Netflix Movie Recommendation System

Team 10

Ng Jing Da 1003496

Lucas Ng 1003478

Tan Yin Ling 1003891

Evan Sidhi 1003691

Table of Contents

INTRODUCTION

Recommendation systems are used in many user facing software, from Spotify to Netflix. In this project, we aim to build a recommendation system through machine learning technology. Namely we will be using the Restricted Boltzmann Machine (RBM). The dataset we are provided contains information about user's opinions of movies in the form of a rating. Given the ratings, our task lies in using the RBM to find out what these users might rate movies that they have not yet watched. This is done by getting the machine to find possible relations between movies that users have watched and other movies.

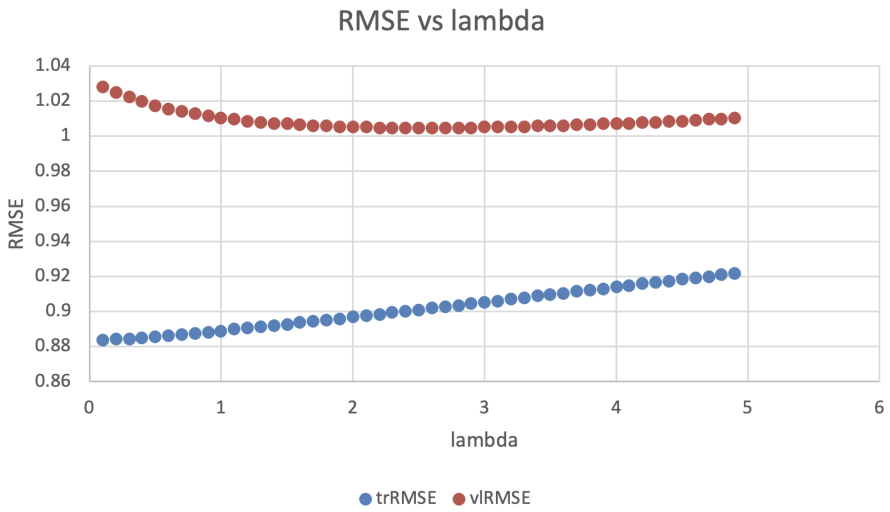
In the context of predicting movie ratings, we will be using Root Mean Squared Error (RMSE) between the predicted and actual rating as the main metric of accuracy in testing our models. The smaller the RMSE, the better the recommendation system. In the context of our data, we have 5 different ratings. This tells us that an RMSE of 0.5 or lower would be ideal as it would result in a change of recommendations for the system. Although RMSE is not ideally what we want to measure, it is a simple and sufficiently effective approximation for the real problem of recommendation.

This report details how we used linear regression to build a dummy model, sharing the validation RMSE. We will then build an RBM, tweaking parameters to obtain an ideal RMSE. Afterwards, we will implement features within the RBM to improve the results on the validation set.

MODELS

Linear Regression

We start out the experiment with a baseline model that we will aim to beat and surpass. The linear regression model is fitted to the data by solving an optimisation problem: to minimise the RMSE. In the linear regression model, the only parameter we could tweak was the regularisation parameter, λ. Increasing the value of λ will increase the penalty that we put on giving weights, discouraging the model from overfitting the values to fit the dataset.



Hence, we will use the results as our basis for comparison to the RBM that we will build.

Linear Regression results

- $\lambda = 2.52$
- Validation RMSE = 1.0047124

RBM

The main bulk of the project was spent on developing the Restricted Boltzmann Machine (RBM). We first began a cursory exploration of how does a RBM model work in collaborative filtering through a series of literature and code reviews, we then moved on to implement a vanilla RBM model and subsequently added more extensions to it in hopes of increasing its prediction accuracy.

Our thought process was to include as many extensions earlier on into the project as this would give us sufficient time to fine tune the hyperparameters. Another consideration is to ensure that the code runs as efficiently as possible to reduce computational time since we are using random searching in our algorithm.

Our group weighed the pros and cons between optimizing the hyperparameters at the beginning and throughout the experiment, we eventually went with the latter as we realised that there could be directed and indirect consequences on how the extensions affect the initial hyper parameters. For example, a higher batch size number could be compensated for a lower learning rate therefore for each model run we included a list of hyperparameter values for our algorithm to random search on, to find the best possible combinations.

IMPROVEMENTS (RBM Model)

Random Searching

We want to search for the best configuration of hyperparameters for our RBM model however we are unsure of how we could optimize the values therefore we decided to adopt a traditional approach of using Random Searching.

This is backed up from our literature review that random searching outperforms grid searching in terms of efficiency when the search space contains more than 4 dimensions. The intent of this Random Searching is to help us identify any possible hidden optimal correlated parameters as we continuously tweaked the extensions of the RBM model.

Below is an implementation of our Random Search, for each hyperparameter there is a list of 5 values where we will randomly choose any one for each model run, the probability of each selection follows that of a normal distribution. This is a tedious and iterative process where our team trial and error with different sequences of values before settling down on a few final versions.

```
K = 5
F_ = [10,11,12,13,14]
batch_size_ = [3, 4, 5, 6, 7]

reg_lambda_ = [0.0010, 0.0011, 0.0012, 0.0013, 0.0014] # Regularisation.
Test from 10^-3 to 10^-2

lr_ = [0.075, 0.08, 0.085, 0.09, 0.095] # LR

# Momentum Parameters
## A medium article recommend the initial momentum beta to be from 0.5 to
0.9
mom_init_ = [0.5, 0.6, 0.7, 0.8, 0.9]
mom_final_ = [0.7, 0.75, 0.80, 0.85, 0.9]

# Bias Learning Rate
vblr_ = [0.01, 0.015, 0.02, 0.025, 0.03]
hblr_ = [0.015, 0.02, 0.025, 0.03, 0.035]
```

Adaptive Learning Rate

Learning rate is a hyperparameter that controls how much to modify the model in response to the estimated error each time the model weights are updated. The simplest implementation would be to make the learning rate smaller once the performance of the model plateaus. On the other hand, the learning rate can be further increased if performance does not improve for a fixed number of training epochs. While conventionally learning rate can be chosen a priori, it would be challenging to choose a good one from the start. This is one of the reasons why the adaptive learning rate method is better in this case. We hope to have a faster convergence of the model than a simple back-propagation with a poorly chosen fixed learning rate (Reed, 1999).

In our case, after rounds of trial and error to find a good initial learning rate, we decided to use a learning rate = 0.1. Afterwards, we have decided to implement three common types of decay on adjusting the adaptive learning rate, namely: time-based decay, step decay and exponential decay.

Time-based decay updates the learning rate with the formula of $learning\ rate\ = \frac{initial\ learning\ rate}{1 + kt}$. This would update the learning rate by a decreasing factor in each epoch. Next, step decay drops the learning rate by a factor every few epochs. A typical way is to drop the learning rate by half every 10 epochs. Last but not least, we also tried implementing another common schedule of updating which is the exponential decay.

The challenge of using learning rate schedules is that these hyperparameters have to be pre-defined by us and their effectiveness depends heavily on the type of model and problem. So after a series of trial and error, our final submission consists only of time-based decay, which we deemed to improve our RMSE the best.

Here is an example of the implementation in our model. The first code block is defined at the top of the code chunk right after we set the for loop for random searching. The second code block is the transformation of the mathematical equation for time-based decay, exponential decay and step decay.

```
  decay_type = "time"
    time_decay_k_grad = grad_lr_init / epochs # Time-based decay
```

```
                # Learning Rate Decay
                # Choose weight learning rate for this epoch
                grad_lr = grad_lr_init
                if decay_type == "time":
                    grad_lr = grad_lr_init / (1 + time_decay_k_grad *
(epoch-1))
```

```
                elif decay_type == "step":
                    grad_lr = grad_lr_init * drop ** np.floor((epoch-1) /
epochs_drop)
                elif decay_type == "exp":
                    grad_lr = grad_lr_init * np.exp(-exp_k * (epoch-1))
                else:
                    print("Error: Wrong Decay Type")
```

Momentum

Before our team dived into momentum, we wanted to find out more about the difference between Stochastic Gradient Descent (SGD) and Batch Gradient Descent (BGD). Batch gradient descent computes the gradient of the cost function with respect to the parameters for the movie dataset whereas stochastic gradient descent performs a parameter update for each training example. The iterative update of stochastic gradient descent helps to reduce redundancy in computation time. Another advantage of stochastic gradient descent is that it allows the model to converge to a potential new global minimum while for batch gradient descent can only converge to a local minimum. With this basic intuitive understanding of the two different gradient descent work, we decided to go with stochastic descent while complementing it with the momentum extension. One limitation of SGD is that in certain scenarios, it oscillates around the ravines, i.e. areas where the contour plot curves much more steeply in one dimension than in another (Ruder, 2020).

In networks with stochastic gradient descent algorithms, implementing momentum accelerates the gradient vectors in the right direction, leading to convergence in a quicker time by damping oscillations. The momentum value increases for dimensions whose gradient points in the same direction and reduces updates for dimensions whose gradients change direction (Ruder, 2020).

Here is an implementation of momentum in our RBM model, we took inspiration from a code found on github (Andrea, 2018).

```
            # EXTENSION: MOMENTUM
            # No momentum for first epoch
            if epoch == 1:
                beta_grad = grad
                beta_vbGrad = vbGrad
                beta_hbGrad = hbGrad
            # Use mom_init for first 5 epochs
##            elif epoch < 5:
```

```
##                 beta_grad = mom_init * beta_grad + grad
##                 beta_vbGrad = mom_init * beta_vbGrad + vbGrad
##                 beta_hbGrad = mom_init * beta_hbGrad + hbGrad
            # Use mom_final for subsequent epochs
            else:
                beta_grad = mom_final * beta_grad + grad
                beta_vbGrad = mom_final * beta_vbGrad + vbGrad
                beta_hbGrad = mom_final * beta_hbGrad + hbGrad

            # Update weight and bias at the end of each batch
            W += beta_grad
            vb += beta_vbGrad
            hb += beta_hbGrad
```

Mini-batches

While it is possible to update the weights after estimating the gradient on a single training case, it would be more efficient to divide the training set into small “mini-batches”. These mini-batches allow matrix multiplication that can be very advantageous with computation using a Graphical Processing Unit (GPU). These data points are split into batches where the model is updated. This also prevents us from having to change our learning rate when the batch size changes.

It is important to note that we updated the visible and hidden bias gradients in the mini-batch loop, but only updated the gradient of the weights in the user loop. Inspired from Geekforgeeks’ article, we implemented this as follows:

```
def create_minibatches(order, batch_size):
      mini_batches = list()
      n_minibatches = len(order) // batch_size
      for i in range(n_minibatches):
            mini_batch = order[i*batch_size:(i+1)*batch_size]
            mini_batches.append(mini_batch)
      if len(order) % batch_size != 0:
            mini_batch = order[(i+1) * batch_size:len(order)]
            mini_batches.append(mini_batch)
      return mini_batches
```

Other Extensions

Methods	Effect
L1 Regularisation - adding the sum of the weights as a cost function to decrease the chances of the model overfitting	We did not see much improvement in the result.
Early Stopping - as the model trains over more and more epochs, the model will overfit. This can be seen as the validation RMSE will start increasing. Early stopping ensures that once the validation RMSE starts increasing, the model stops training.	For results where the validation RMSE was constant, this did not yield much result.
Biases - due to each user having their own specific set of preferences when rating movies, this would affect the RBM accuracy in predicting ratings if we did not add a separate bias for each user. We tried implementing a bias value for each visible unit of the different values. This was inspired by a few literature review and code we found online (Hinto & Ruslan, 2021)	We noticed a slight improvement in our RMSE value ~0.04 however subsequent attempts to modify the equation did not further improve the result. We are unsure if the biases implementation was done correctly.
Dropout Layer Regularisation - this is an addition to the L1 regularisation parameter that we have added above. The intent of this is to increase the robustness of the network by allowing it to randomly deactivate neurons to remove any simple dependencies between the neurons without any triggering error. In essence, this gives rise to more variants of the RBM model as the units respective weights are not updated.	Most of the references we found online use built in libraries to perform this function, however our group decided to code it from scratch. This resulted in our model to underperform as the RMSE value went upwards to 1.21-1.22. Eventually, we decided to not implement this.

Most of the code implementation of the mentioned extensions can be found in the mainRBM.py file. We created new functions where necessary in the rbm.py file.

RESULTS

Week	Test Submission RMSE	Extensions Used
Week 9	1.1354	Random Searching, Adaptive Learning Rate, Momentum, Mini-batches, L1 Regularisation, Early Stopping, Biases

Week 10-11	1.1346	Random Searching, Adaptive Learning Rate, Momentum, Mini-batches, L1 Regularisation, Early Stopping, Biases, **Dropout Layer Regularisation** **Hyperparameter Tuning**
Week 12	1.1507	Random Searching, Adaptive Learning Rate, **Momentum**, Mini-batches, L1 Regularisation, Early Stopping, **Biases** Adjusted momentum and biases computation, did not work out as intended.

CONCLUSION

Learning Takeaways

If we had the luxury of time on the project, we would like to explore how we can tweak the objective function (in this case, minimising the RMSE) to something that simulates closer to the recommendation system's objective. This might take the form of rounding off values to the closest integer to simulate a user's pick.

We also would explore a Bayesian optimisation approach on top of the simple random searching mentioned. The challenging aspect of this project lies in digesting the widely available academic reports on this field and translating that knowledge into code. In practice and theory, the RBM model should outperform the Linear Regression model and one possible reason we did not manage to achieve a similar performance is our implementation of the biases extension. Another final improvement we would consider is the adding of Gaussian Visible Units to our model instead of Binary Units.

This project has allowed our team to have a better appreciation of the large and complex algorithms that are powering behind the scenes in our daily lives and that it is important to develop both technical and theoretical understanding.

REFERENCES

Sebastian Ruder. “An Overview of Gradient Descent Optimization Algorithms.” Sebastian Ruder, Sebastian Ruder, 20 Mar. 2020, ruder.io/optimizing-gradient-descent/index.html#stochasticgradientdescent\

ML: Mini-batch gradient descent with Python. (2019, January 23). Retrieved April 30, 2021, from https://www.geeksforgeeks.org/ml-mini-batch-gradient-descent-with-python/

Andrea-V. (n.d.). Andrea-V/Restricted-Boltzmann-Machine. Retrieved April 30, 2021, from https://github.com/Andrea-V/Restricted-Boltzmann-Machine/blob/master/rbm_train.m

Salakhutdinov, R., & Hinton, G. (n.d.). Training a deep autoencoder or a classifier on MNIST digits. Retrieved April 30, 2021, from http://www.cs.toronto.edu/~hinton/MatlabForSciencePaper.html

Oppermann, A. (2020, August 11). Regularization in deep learning - l1, l2, and dropout. Retrieved April 30, 2021, from https://towardsdatascience.com/regularization-in-deep-learning-l1-l2-and-dropout-377e75acc036

Adam PAdam P 8122 bronze badges, & Edward NewellEdward Newell 21133 silver badges66 bronze badges. (1963, November 01). Updating bias with rbms (restricted boltzmann machines). Retrieved April 30, 2021, from https://stats.stackexchange.com/questions/139138/updating-bias-with-rbms-restricted-boltzmann-machines

Rbm - linear to binary layer. (n.d.). Retrieved April 30, 2021, from https://www.mathworks.com/matlabcentral/answers/215197-rbm-linear-to-binary-layer

Reed, R. D., & Marks, R. J. (1999). *Neural smithing: Supervised learning in feedforward artificial neural networks*. Boulder, CO: NetLibrary.