Networked Life Project Report

Group HAO: Zhang Hao (1000899), Zhao Juan (1000918), Guo Ziqi (1000905)

Introduction

In this report, we will detail the two models we implemented on the Netflix dataset, linear regression and the restricted Boltzmann machines. For each of the two models, we will explain the thinking behind the extensions we added to the model, and also demonstrate how these extensions have improved our model's performance.

Linear Regression

In unregularized linear regression, we use *formula (1)* to obtain the optimal set of **b** which minimizes RMSE.

$$b^* = (A^T A)^{-1} A^T c \quad (1)$$

For regularized linear regression, we introduced a parameter λ, to penalize non-zero parameters. Thus, the optimal set of *b* is obtained using *formula (2)*:

$$b^* = (A^T A + \lambda * I)^{-1} * A^T * c \quad (2)$$

We tune the value of λ to achieve the minimal RMSE in training set and validation set, graph below show how validation RMSE evolves as λ changes:

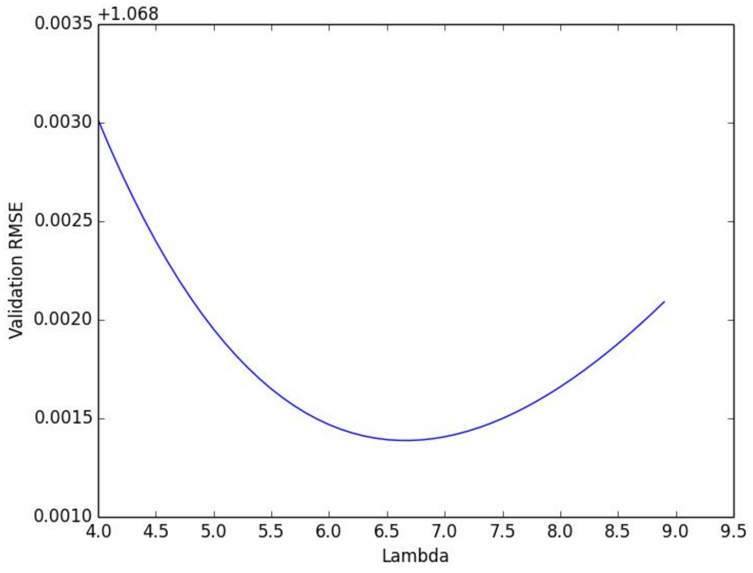

Figure 1. Validation RMSE V.S. Lamda λ

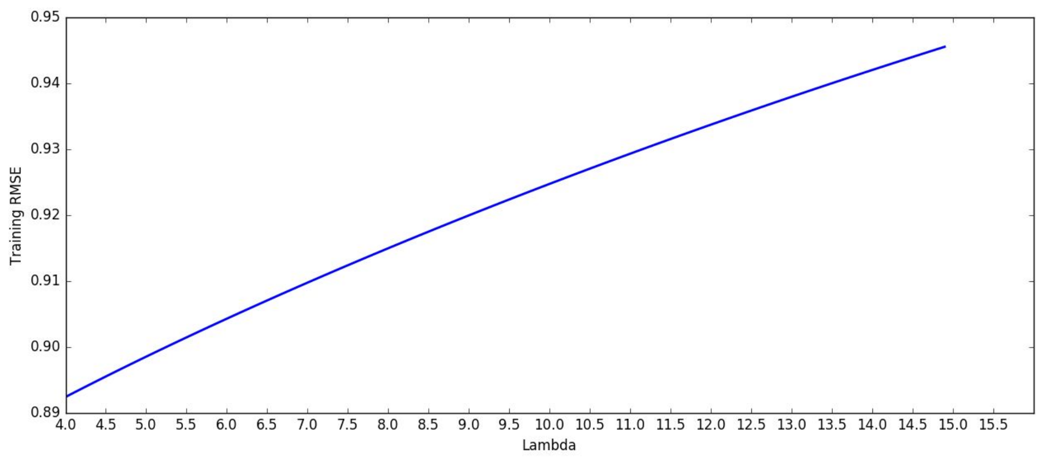

Figure 2. Training RMSE V.S λ

We choose $\lambda = 6.7$ since it gives the lowest validation RMSE of 1.069.

Restricted Boltzmann Machines

1. Momentum

To stabilize the weights in each progressive epoch, we added the momentum property. Like the principle in physics that any object with high inertia is likely to preserve its momentum, we applied the same thing to weights in RBM. Therefore, in each epoch, new_weights = momentum * old_weights + (1 – momentum) * new_prediction, instead of new_weights = new_prediction, is used. Momentum here is a parameter that ranges from 0 to 1, which determines the fraction of the old weights preserved. If we set the momentum too large, it will result in an extremely steady gradient update and we won't be able to train the dataset well; and if we set the momentum too small, the gradient update will be oscillating. Hence, it is crucial to find the optimal value for momentum.

Fixing other parameters (F=7, epsilon=0.001, epoch=30, weightcost=0.001), and adding a momentum of 0.5, we obtained the following graph:

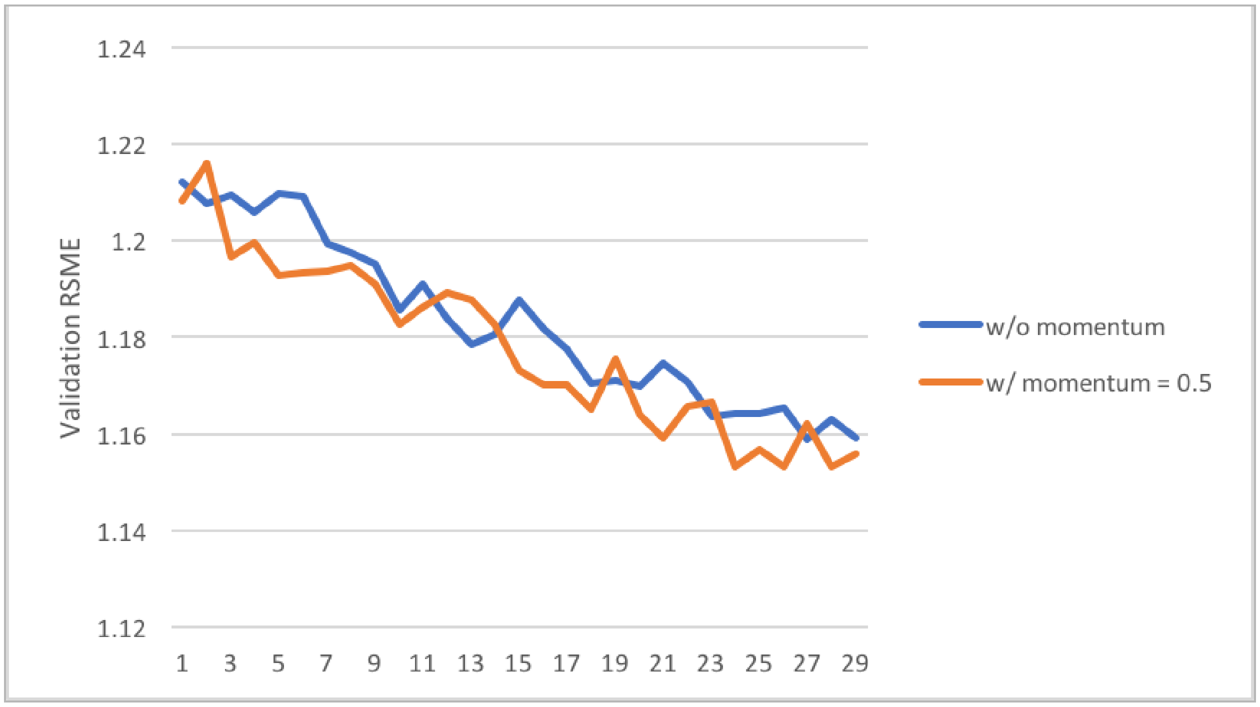

Figure 3. Validation RMSE with and without momentum

From the graph, we can observe a decrease in validation RMSE if momentum = 0.5 applied.

2. Adaptive learning rates

We started the project with a fixed learning rate epsilon = 0.01. If the learning rate is set too high, the algorithm can oscillate and become unstable. If the learning rate is too small, the algorithm takes too long to converge. It is very difficult to find an optimum fix learning rate that make the gradient update both stable and converge in a short time. To fix this problem, we need to find some method that decreases the learning rate as the algorithm enters next iteration. . We decided to set the adaptive learning rate to base_learning_rate / epoch. In this way, the learning rate will start with base_learning_rate and it keeps decreasing in each epoch.

Fixing other parameters (F=7, epsilon=0.001, epoch=30, weightcost=0.001, momentum=0.5), and run with fixed learning rate and adaptive learning rate, we obtained the following graph:

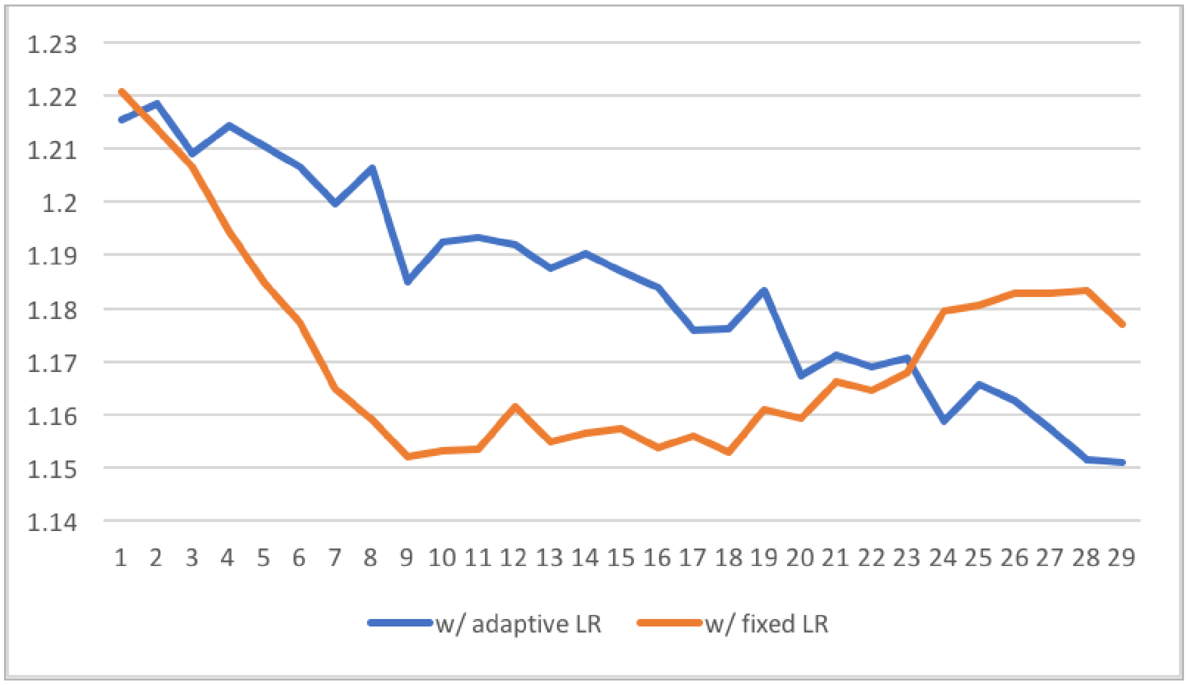

Figure 4. Validation RMSE with or without adaptive learning rate

The graph with fixed learning rate is an example of setting learning rate too large: The validation RMSE drops very fast and then it started oscillating. We can conclude from the graph that with the application of adaptive learning rate, the problem of overfitting is alleviated.

3. Early stopping

With parameters (F=7, epsilon=0.001, epoch=100, weightcost=0.001, momentum=0.5), we can get the following graph.

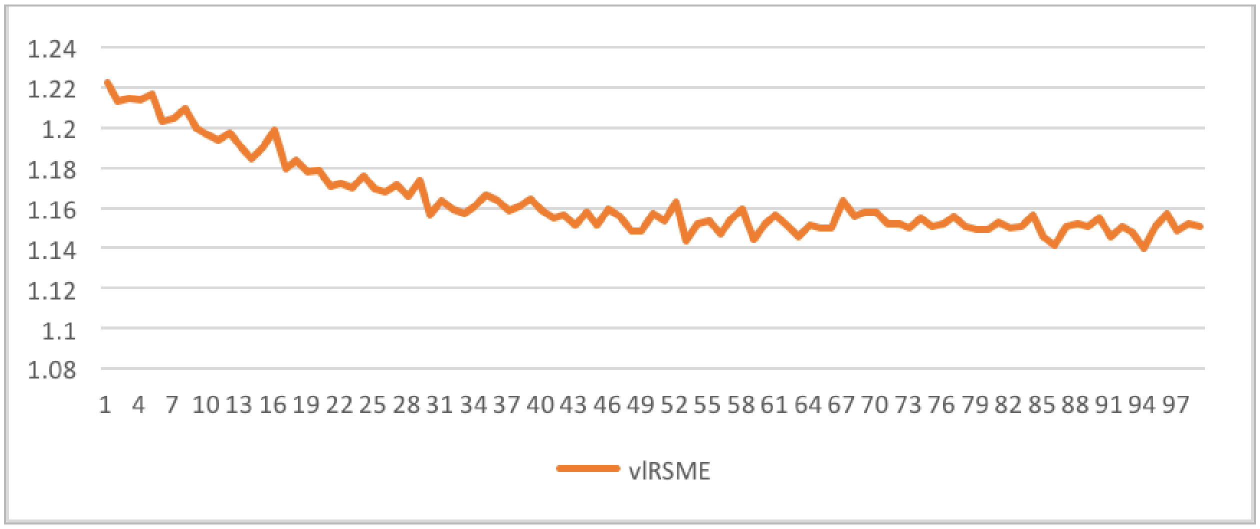

Figure 5. Evolution of validation RMSE

The validation RMSE tends to converge after epoch 50, however, it still oscillates in a small interval. Also, we noted that the iterating through more epochs does not mean you can get the best RSME. Therefore, we record down the best RMSE during training and use the weights with best RSME to predict the test data.

4. Regularization

Similar to the regularization in linear regression, we introduce a parameter *weightcost* to penalize the non-zero weights. As shown below in *formula (3)*, we multiply *weightcost* with the L2 norm of the weights W, and minus it with the original gradient..

*grad = momentum * grad + (1 - momentum) * gradientLearningRate * (*
*(posprods - negprods) / trStats['n_users'] - weightcost * np.linalg.norm(temp))   (3)*

However, regularization doesn’t help to reduce the RSME significantly based on our testing.

To penalize the weights that grow too large, we are using Euclidean norm (L2 norm) to measure the size of the weights W.

So in mainRBM, where we update the gradient, not only we need to consider the difference between posprods and negprods, but also the Euclidean norm of the weights. A suitable function in Python is the linalg.norm under Numpy package. We implement this by deducting the product of a cost coefficient we set and the norm from the formula. As a result, we have a new parameter that we can tune here, which is the weightcost associated with the norm. The larger it is, the more heavily the norm of the weights will be penalized.

Fixing other parameters (F=7, epsilon=0.001, epoch=30, momentum=0.3), and adding a weightcost of 0.001, we obtained the following graph:

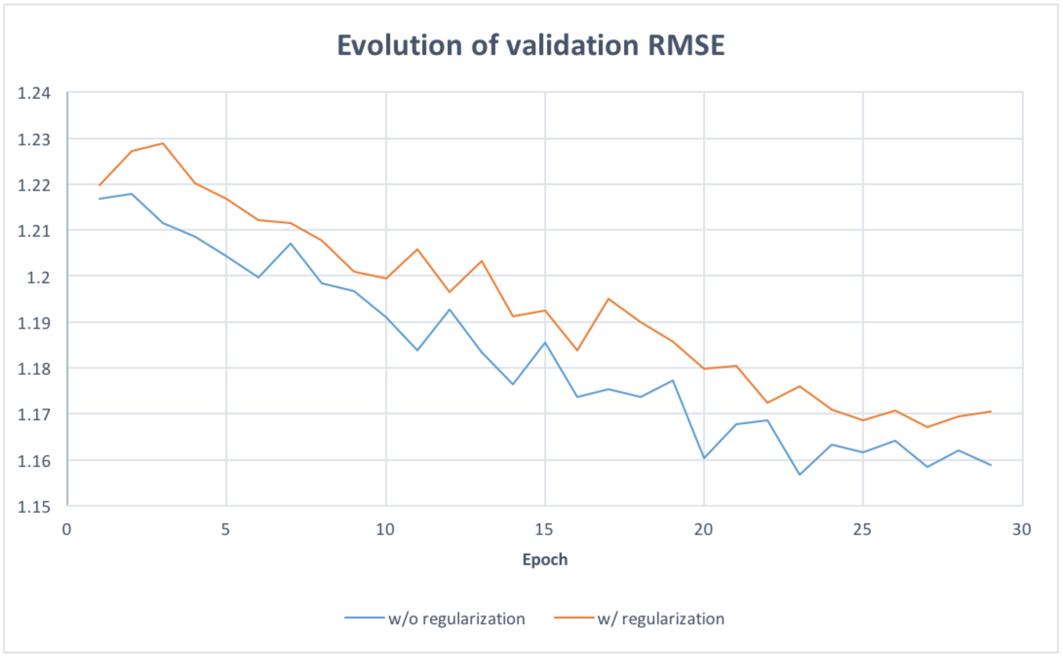

Figure 6. Validation RMSE with or without regularization

We can see that with regularization, it seems that the validation RMSE decreases at about the same rate as without regularization. But the overall level of RMSE is even higher comparatively. This is an indication of under-fitting. Therefore, adding a penalty for the size of weights does not always help with the model.

5. Mini batch

In the given setup in mainRBM, in each epoch all the users will be shuffled into a random order and all the users will be gone through according to this order. The gradient increase of each user is discounted by the total number of users, so that essentially the function is taking the average of all the gradient increases that are generated from all users.

A problem with this approach is whenever a user's ratings are examined, his gradient increase will immediately be added back into W, which keeps track of all the weights. This means when examining the next user, we are already using the updated weights. This updating rule will make the training process very sensitive to each user's ratings, so it's not a very efficient approach to train the weights.

Therefore, in the mini batch setup, we partition all the users into batches of size B. We will only update the weights after going through one batch of users, as opposed to the almost real-time feedback in the original case. In this case, each user's deviation will not have a huge and immediate impact on the weights, but will be moderated by other users that are also in his batch.

The parameter that we can optimize is the batch size B. A larger batch size will take more samples into account and might help with overfitting.

Fixing other parameters (F=7, epsilon=0.001, epoch=30, momentum=0.3, weightcost=0.001) and choosing a batch size of B=10, we obtained the following graph:

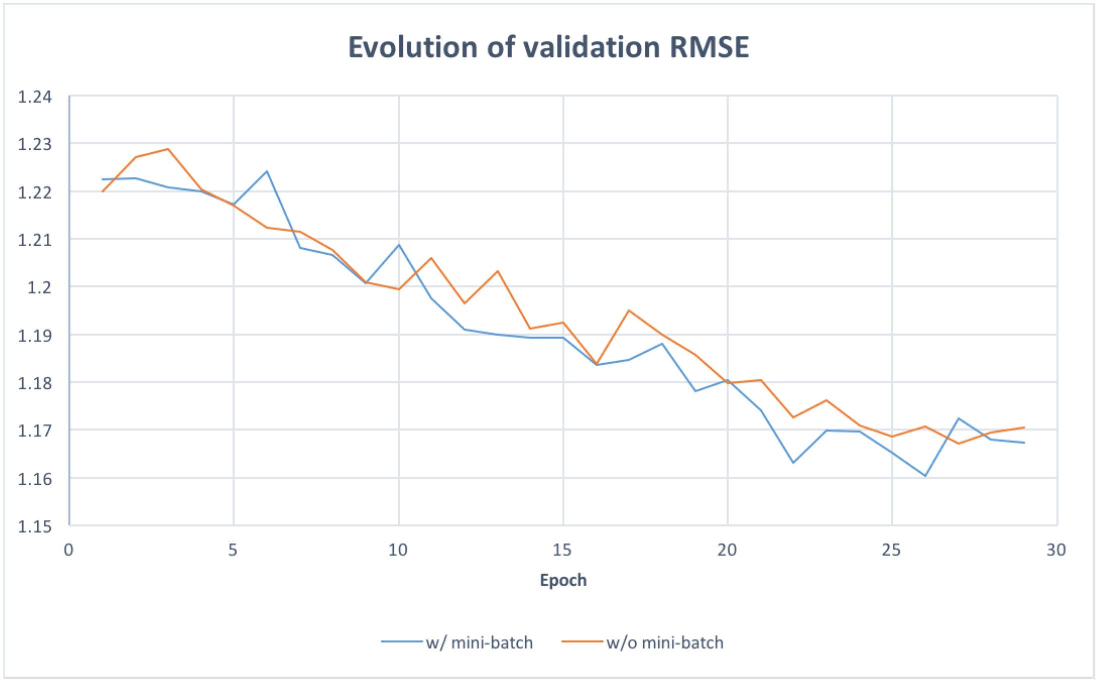

Figure 7. Validation RMSE with or without mini-batch

From the graph we can see that adding the mini-batch implementation seem to smoothen out the training process a little bit, as the update of weights is decided by 20 training examples at a time, instead of one. However, the batch-size B can also be optimized through a grid search that will implemented at the end.

6. Biases

We also went on to include the bias units in our restricted Boltzmann machine. In the basic model, the predicted ratings are calculated from two main steps, forward and backward propagation. The forward propagation is basically a product sum of the binary input to the visible units and the weights between each pair of visible and hidden unit. By introducing hidden biases to the model, we are adding a lump sum to the calculation of each activation of hidden unit that would hopefully capture the distinctiveness of this latent factor after training. Likewise, we can also add visible biases to improve the results of backward propagation.

The way we train biases is by adding an additional visible and hidden unit to the model. The additional visible unit will receive 5 binary inputs, just like all the other visible units. If all 5 inputs always have value 1, they will essentially add their corresponding weights to the activation function of each hidden unit. We do the same for the hidden units. So we make sure that whenever forward or backward propagation is used, the last row (or element, in the case of hidden unit) is all set to 1, representing the activation of bias units. In this way, we avoid updating biases' weights separately in the gradient descent, but rather let the model train the weights W altogether.

Fixing other parameters (F=7, epsilon=0.001, epoch=30, momentum=0.3, weightcost=0.001, B=10) and implementing the bias extension, we obtained the following graph:

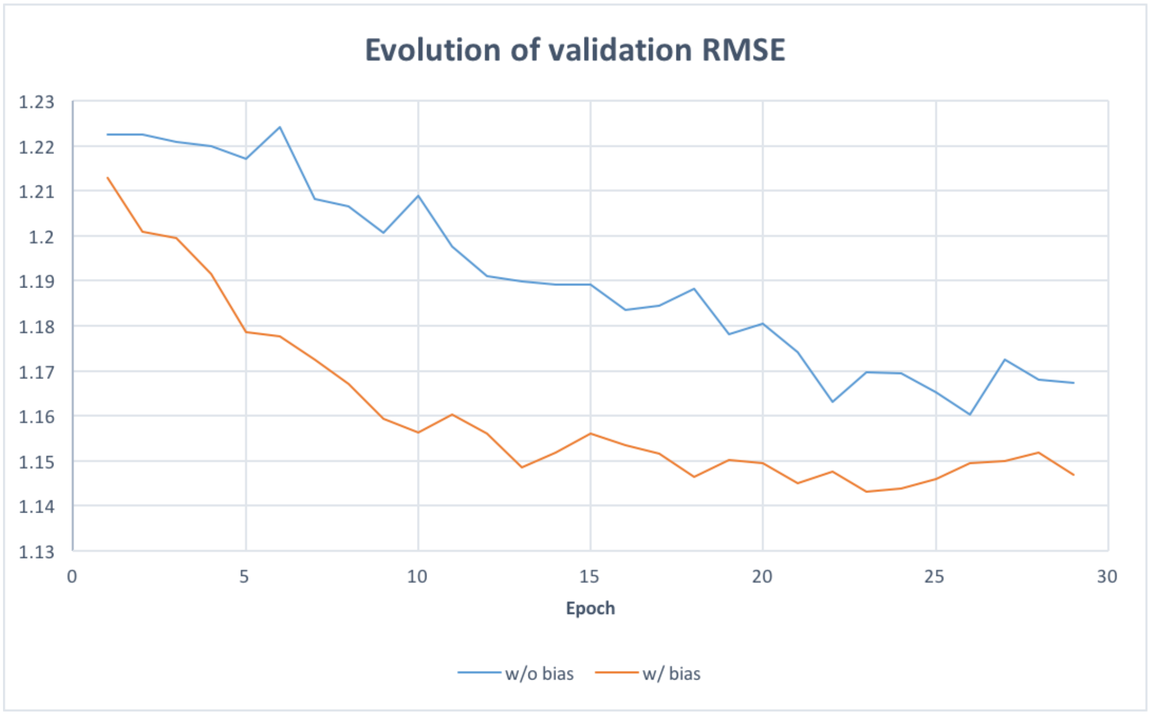

Figure 8. Validation RMSE with or without bias

It is quite obvious that adding the bias nodes into the model leads to a pretty significant performance leap. Especially at the beginning, the red curve with bias dips much faster than the curve without bias. However, as epoch increases, the improvement slows down, which suggests that this set of parameters has some room for improvement.

7. Final model

After implementing all the extensions of RBM and playing with different hyperparameters, we selected a range for each hyperparameter, in which we think the best combination lies. Specifically, we configured the range of values as follows:

- F: 4, 6, 8, 10, 12, 14
- Epsilon: 0.005, 0.01, 0.02
- B: 10, 20
- Weightcost: 0.0005, 0.001
- Momentum: 0.3, 0.5

We set the epoch to be 50, as no matter how much we slow down the learning rate, the evolution of validation RMSE tends to converge before an epoch of 50.

To find the best combination of hyperparameters within the aforementioned range, we adopted a grid search approach, which is to iterate through all combinations and save their respective errors and parameters. If the best parameter is at the limits of the range, we moved the grid search window accordingly. Eventually, we selected the combination with the lowest validation RMSE and predict the whole set using the weights associated with it.

In the final model we selected, we used the following hyperparameters:

- F = 6
- Epsilon = 0.01
- B = 20
- Weightcost = 0.001
- Momentum = 0.3

This model gives us a validation RMSE of 1.128.

Conclusion

To sum up, after trying different extensions of RBM and optimising the hyperparameters through grid search, the model is still not as accurate as a linear regression with regularization. A takeaway here is that sometimes a simple linear model is able to model certain datasets nicely, so we should always choose the model that befits the data. Eventually, we chose the linear regression trained on both the training and validation set to get the most learning.

The RBM, on the other hand, can be further improved. One possible improvement is to design the adaptive learning rate so that it responds to the evolution of error or the norm of the weights, instead of simply slowing down with epoch.