Beyond the Truth: Data Augmentation via Latent Features in Recommender Systems

Benjamin Steiner
Department of Mathematics
The London School of Economics and Political Science (LSE)
London, England
B.Steiner1@lse.ac.uk

***Abstract*—In this paper we propose a new method of data augmentation in cases of missing values in our input. By adding Gaussian noise to the latent features, we are able to oversample our dataset to improve performance and reduce overfitting of our autoencoder. To validate our hypothesis, we provide a proof of concept through an application to a Recommender System (RS). We find that the data augmentation leads to slightly better results, but we observe the greatest improvement in the early stages of the learning process. We attribute this to the data augmentation providing a form of regularisation to prevent overfitting.**

***Index Terms*—Latent Features, Recommender Systems, Autoencoders**

I. Introduction

Neural Networks have the ability to perform remarkably well in many prediction tasks, however, they can often be reliant on having a large dataset for training. This may not always be feasible, so we must come up with ways to prevent overfitting even with limited data. By introducing some regularisation into the training process, we are able to improve generalisability to other settings.

The most common neural network architecture to learn the patterns of data for a RS is the *autoencoder*. The goal of the autoencoder is to predict the ratings of the all items - including the ones missing from the input.

Inspired by [1], we propose a similar method of adding Gaussian noise to the input. By adding noise, we impose a form of regularisation that can both improve performance on test data and also reduce overfitting.

A. The Dataset

During our testing phase, we use the MovieLens 100k dataset, which serves as a research-quality dataset for movie recommendations. Our primary objective is to predict the ratings that a user would assign to each item, enabling us to identify items that the user is likely to appreciate or might not have previously considered. By achieving this goal, we aim to enhance personalized movie recommendations and provide users with tailored suggestions for an enhanced experience.

The dataset contains $99,946$ ratings across 9,724 movies for 610 users, exhibiting a *sparsity* rate of 98.3%. This means that, on average, each user has 163 ratings given, and 9561 missing values.

B. Problem Setting

Recommender Systems provide a very interesting example of data, whereby the vast majority of our inputs will be *missing values*. This is because the number of options available to a consumer may far exceed the number of explicit ratings a user has given. To represent these missing values, we commonly use *0* in our input. However, we could equally use any placeholder value. Therefore, we are unable to simply add the noise to our input vector. The novelty of our problem comes from finding another representation of our input vector to add noise to. As we will see later in Section III, autoencoders provide a way to obtain a representation without missing values.

II. Recommender Systems

With so many choices available to consumers nowadays, we require a method to filter for the best items. Recommender systems leverage a user's past history and the actions of similar users to suggest items that a user will like the most.

In the subsequent sections we give a brief overview of the existing literature of recommender systems. For a more detailed explanation, we refer the reader to [2].

A. Definitions

More formally, let $u \in U$ be a user, and associated with each user are the ratings r_u. We denote the rating given by user u to item i by r_{ui}. For each user, we let I_u be the set of indices for which user u has a non-zero rating. This last part is particularly important in relation to *masked values* when we expand on it further in Section III-B.

B. Model Evaluation

During the training process, we aim to learn a function $f : \mathbb{R}^n \to \mathbb{R}^n$ that captures meaningful patterns and relationships within the data. In order to evaluate our model, we need a way to measure the performance capabilities of the model and accuracy of our recommendations. While there are many such measures, in this paper we use *Mean-Squared Error* as it provides a continuous and smooth loss function, allowing for optimization using gradient-based methods.

Definition 1: Let U be the set of users and r_u be the ratings given by user u. The *masked mean-squared error* is a loss function $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$ defined to be:

$$\mathcal{L}(r_{ui}) = \frac{1}{|U|} \sum_{u \in U} \sum_{i \in I_u} (r_{ui} - f(r_u)_i)^2.$$

III. Autoencoders

In this paper, we adopt an autoencoder, a type of neural network, to train our model for the given task. This approach offers significant advantages as it enables us to harness the strengths of different solutions to the problem. Leveraging the power of deep learning, our model demonstrates high performance in prediction tasks. Simultaneously, we benefit from the interpretability of collaborative filtering, which is known to be one of the most transparent methods for generating recommendations.

A. Network Architecture

In [3], they suggested the two-network structure used in an autoencoder: the *encoder* and the *decoder*. First, we pass the input through the encoder which reduces the dimension at each layer. This process aims to learn a lower-dimensional representation of our input. We call this representation the *latent features*. We then pass these latent features through the decoder to re-construct the input and predict ratings for all of the items. In Figure 1 we show an example of an autoencoder.

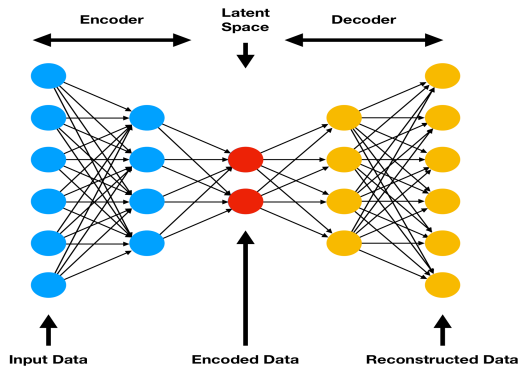

Fig. 1. An example 6-4-2-4-6 autoencoder architecture.

B. Masking

Earlier, we mentioned *masking* - the process of "ignoring" certain items during the loss calculation. This is particularly important when dealing with sparse data. During the loss calculation, we only want to determine the loss between items with a rating in the input. Failure to do so can result in the trained model being close to the *null model*. That is, the prediction for most items would be close to zero. We call the step of ignoring the missing values in the output *masking*.

IV. Methodology

In this section, we describe the process of data augmentation and our final network architecture. Throughout this section, we will refer to the model trained on the original dataset as the *regular autoencoder*, and the model trained on the augmented dataset as the *augmented autoencoder*.

A. The Data Augmentation Procedure

One of the key issues this method aims to address is the problem of overfitting. Through the addition of noise, we will decrease the variance of the training process, leading to a more stable and reliable model. The introduction of carefully controlled noise during training serves as a regularisation technique, preventing the model from becoming overly sensitive to individual data points or noise in the training data.

In order to augment our dataset, we must create copies of our original data and perform some form of transformation to them. To extract the latent features, we pass the inputs through the autoencoder until we reach the latent space - essentially using the trained autoencoder as a proxy for the truth. From here, we must now pick suitable values for two hyperparameters: the *standard deviation of the noise*, and *fraction of features* we add noise to. This process is essential to optimise the performance of the augmented autoencoder. We discuss in detail our findings of suitable values in Section IV-C. Finally, we decode these noisy latent features to attain our augmented dataset.

B. Final Architecture

We encountered several trade-offs that required careful consideration: achieving high test performance, avoiding overfitting, and ensuring computational efficiency. For our final model, we settled on a network architecture with 5 hidden layers and $9724 - 200 - 50 - 20 - 50 - 200 - 9724$ neurons in each layer. This choice was driven by the need to strike a balance between accurate predictions and efficient learning. By adopting this architecture, we could effectively capture complex patterns without compromising the model's generalisability. Although deeper networks often yield improved predictions, we recognised that with our relatively small dataset, an excessively deep network would be prone to overfitting, necessitating a more cautious approach.

An important question that guided the architecture of the network was how to interpret the data augmentation procedure. In [1], by adding noise to the colour channels of the image, they are altering the intensity of certain pixels. However, in our case it is not quite so simple as we are adding noise to the latent features - not the explicit ratings given by a user. In the case of movie data such as ours, the latent features represent aspects of each movie, for example, genre, popularity, amount of CGI, or even soundtrack intensity. In general, a latent space with fewer features will make the features more interpretable but may lead to *underfitting* the model, as it might struggle to capture the full complexity and variations present in the data.

C. The Hyperparameters

In addition to the "standard" hyperparameters used in deep learning such as *learning rate*, *number of epochs*, *dropout*, *number of layers/nuerons*, and *batch size*, the data augmentation procedure introduces three new hyperparameters.

Firstly, we must decide how much noise to add: the *standard deviation of the noise*. Crucially, this determines how similar the noisy features are to the original latent features. We aim to strike a balance between generating distinct examples while the new features still being representative of the distribution of our data.

We must also decide on the *total data augmentation*, i.e by how much we increase the size of the dataset. Whether we believe the data to be of good quality should guide us in the size of the augmentation and so the value we pick is highly contextual.

Finally, we can also pick the number of features to which we add the noise to, that is, the *fraction of noisy features*. This parameters is highly exclusive to our setting of using the latent features to add noise. In scenarios with no missing values, without a deep understanding of the data collection process and possible biases, we should most likely add noise to all of the input features.

1) Learning Rate: while many values for the learning rate will find the optimal solution, we observe divergence away from this solution once it has been found. Therefore, we pick learning rate of 0.001 which is able to balance finding a solution in a reasonable number of epochs, but without this divergence.

2) Number of Epochs: we find that a solution can be found in relatively few epochs.For good measure, we use 50 epochs, although as will be seen later in Figure 2, there is a severe flattening of the learning process after about 30 epochs.

3) Dropout: while the data augmentation procedure provides an implicit regularisation, we utilise another form of regularisation through dropout [4]. In order to mitigate too much overfitting, we implement a dropout rate of 0.2 in each Fully Connected layer.

4) Number of layers/neurons: as one of the most important features of the model, we earlier dedicated Section IV-B to this discussion. In short, we must choose an architecture that is complex enough to learn meaningful patterns, without completely overfitting the model.

5) Batch Size: with a smaller dataset, the training process is less effective in the early stages with a mini-batch size that is too large. This is likely due to the fact that with large mini-batches we do not perform enough updates to learn the patterns of the data. However, we must still balance the high possibility of overfitting so we choose a neutral mini-batch of 12.

D. Computational Equity

Perhaps the most important difference is that the data augmentation procedure produces a dataset that is larger than the original one, and also has a different sparsity.

In order for both models to perform the same number of updates in each epoch, we must increase the size of the mini-batches in the augmented autoencoder. We are able to specify the exact relationship between the two in the following theorem.

Theorem 1: Let the models 1 and 2 denote the regular autoencoder and augmented autoencoder, respectively. We denote the size of the dataset by n, mini-batch size by b, and the number of batches by B.
In order to equate the number of batches used during each epoch, we must pick the batch size, b_2, according to the function $g : \mathbb{R} \to \mathbb{R}$ defined by

$$g(n_1, n_2, b_1) = \lceil \frac{(1+c)n_1}{\lceil \frac{n_1}{b_1} \rceil} \rceil,$$

where $c \in [0, 1]$ denotes the fraction by which n_2 is larger than n_1.

Proof: During each epoch, there are $B = \lceil \frac{n}{b} \rceil$ updates (which is also the total number of mini-batches). We want to find a relationship such that $B_1 = B_2$. However, from above we know that $B_1 = B_2$ is equivalent to

$$\lceil \frac{n_1}{b_1} \rceil = \lceil \frac{n_2}{b_2} \rceil. \tag{1}$$

We also know that $n_2 = (1+c)n_1$ for $c \in [0, 1]$, so we can re-write Equation (1) as:

$$\lceil \frac{n_1}{b_1} \rceil = \lceil \frac{(1+c)n_1}{b_2} \rceil. \tag{2}$$

Re-arranging, we obtain

$$b_2 = \lceil \frac{(1+c)n_1}{\lceil \frac{n_1}{b_1} \rceil} \rceil, \tag{3}$$

and this completes the proof. ■

V. Results

We found that our method does in fact improve performance in key areas. In particular, it provides a 0.23% reduction in test loss. While small, there is sufficient evidence to believe that this difference should be a positive number. In all 50 simulations, we observe a lower test loss with the augmented dataset, indicating that the augmentation technique consistently improves the model's performance. This outcome suggests that the augmented data provides valuable additional information to the model during training, enabling it to better understand and capture the underlying patterns in the data.

TABLE I
Data Augmentation Comparison

Table Head	***Augmented Dataset***	***No Augmentation***
Training Data Size	536	488
Sparsity (%)	88.5	98.3
Batch Size	14	12
Learning Rate	0.001	0.001
Dropout	0.2	0.2
Stand. Dev. Noise	0.1	N/A
Frac. Noisy Features	0.5	N/A

A. Speed of Learning

We observe the biggest improvement from this method in the speed of learning. Although the final performance of the two models is roughly the same, the model trained on the augmented dataset performs much better in the early stages of learning.

We hypothesise that since we are using a relatively small dataset, even using a mini-batch size of just 12 means that there are not enough updates during each epoch to learn an effective generalisation of the data. Therefore, we attribute the improvement in early-stage performance mainly to the implicit regularisation and decreased sparsity that the data augmentation procedure provides. We show this feature below in Figure 2.

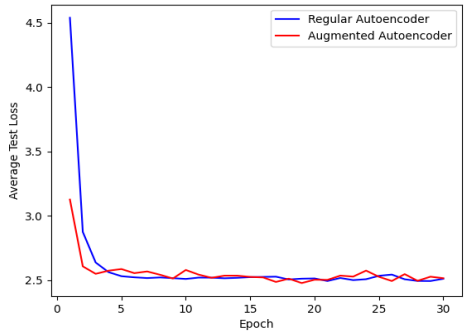

Fig. 2. A comparison of the speed of learning in the first 10 epochs for the two models.

B. Data Sparsity

We must note that the two datasets have different sparsities as a result of passing the noisy latent features through the decoder during the data augmentation procedure. Upsampling by the 10% leaves us with an augmented dataset with average sparsity of 88.5%. We attribute a large amount of the benefits from the augmentation due to this decrease in sparsity. Otherwise, the data augmentation is not much better than adding noisy copies of the original data. However, we must also note that we have only added X additional samples, showing how efficient this procedure is.

VI. Conclusion

Overall, the observed reduction in test loss with the augmented dataset (particularly in the early stages) not only highlights the effectiveness of data augmentation for improving generalization but also indicates its potential in enhancing the model's ability to handle new, unseen individuals and promoting fairness by reducing biases in the user's ratings.

A. Our Rationale

One possible explanation for this improvement is that data augmentation can effectively capture "new" individuals or samples that were not present in the original dataset. By introducing noisy variations to the existing data, the augmented dataset enriches the model's training set with a broader range of instances, allowing it to learn more diverse representations. Consequently, the model becomes more robust and capable of generalizing to previously unseen individuals, leading to a reduction in test loss.

Moreover, data augmentation can also play a crucial role in mitigating biases that might be present in the user's ratings or data collection process. Biases in data can often lead to skewed and inaccurate model predictions. By augmenting the dataset with transformed versions of the original samples, the augmentation process helps to balance the representation of different classes or categories in the data.

B. Further Work

In this paper, we give a proof-of-concept of the effectiveness of the data augmentation procedure to improve performance and generalisation to unseen data. One obvious question to explore further is how well this procedure generalises to other datasets. Examining the use of this procedure will help determine the feasibility for future use. Furthermore, it may enhance the interpretability of the latent space and help us gain deeper insights in different contexts.

Another course of work would be to find the optimal combination of the many hyperparameters involved using a grid search. While we did spend a fair amount of time tuning the hyperparameters, there is surely a better combination than the one presented in this paper.

C. Codebase

All of our code was built using Python, allowing us to leverage PyTorch - a powerful package for deep learning. We recomend using Google Colab to use the GPU hardware accelerator as the training process for complicated networks can be slow. The code can be found on a public GitHub repository here: https://github.com/BSteiner1.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in Advances in Neural Information Processing Systems (NeurIPS), pp. 1097–1105, 2012.

[2] D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich, "Recommender Systems: An Introduction," Cambridge University Press, 2nd edition, 2021.

[3] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "Auto-Associative Memory," Biological Cybernetics, vol. 52, no. 4, pp. 211-220, April 1985.

[4] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, I. and R. R. Salakhutdinov "Improving neural networks by preventing co-adaptation of feature detectors." arXiv preprint arXiv:1207.0580, 2012.