University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Albert Cieślak
Student no. 370756

Michał Filipiuk
Student no. 385423

Frederic Grabowski
Student no. 382434

Radosław Rowicki
Student no. 386088

Variational Autoencoder for Collaborative Filtering – Implementation and Performance Optimization

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
Grzegorz Grudziński
Faculty of Mathematics, Informatics, and Mechanics

June 2019

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Bachelor of Computer Science.

Date Supervisor's signature

Authors' statements

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Authors' signatures

Abstract

Deep neural networks have significantly pushed state-of-the-art solutions' abilities in a wide variety of applications. Typically, deeper architectures are able to achieve better results, increasing model complexity and consequently computational resources needed for training. A number of technologies, e.g. mixed precision or multi-GPU training, have been proposed to combat this issue in fields like computer vision or natural language processing. In this paper, in collaboration with NVIDIA, we investigate practical performance gains achieved on the example of variational autoencoders in collaborative filtering [Dawen Liang, Rahul G. Krishnan, Matthew D. Hoffman and Tony Jebara. Variational Autoencoders for Collaborative Filtering. WWW'18]. This network differs vastly from most of currently analyzed architectures as they were much more computationally expensive and, in result, less prone to being bottlenecked by CPU operations or data transfers. Those obstructions could easily diminish effects of calculational improvements. To address these issues, we take advantage of fact that data fed to our model represents sparse user-item interactions – usage of sparse instead of dense matrices allows us to reduce the existing data transfer bottleneck and achieve a 12x speed-up in comparison with implementation presented in paper. Furthermore, we introduce mixed precision and multi-GPU training to our model, which combined with all other improvements obtain over 40x speed-up when trained on one GPU and almost 150x speed-up on four GPUs.

Keywords

recommender systems, collaborative filtering, variational autoencoder, multi-GPU, mixed precision

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Computer Science

Subject classification

Computer systems organization; Neural networks
Computing methodologies; Ranking

Tytuł pracy w języku polskim

Zastosowanie architektury Variational Autoencoder w problemie Collaborative Filtering - implementacja i optymalizacja wydajnościowa

Contents

Chapter 1

Introduction

Nowadays, people are overwhelmed by the amount of data available on the Internet. Every minute about 400 hours of video is uploaded to YouTube [23]. Every day 500 million tweets are posted on Twitter [20]. Times when you could read through everything to find something interesting on the Web are gone. Recommender systems allow us to sift through this digital mess.

1.1. Recommender systems

A recommender system or a recommendation system is a system whose objective is the prediction of user rating or preference about a certain product. They can be used to solve various problems: generating custom playlists in streaming services, suggesting the next movie to watch or finding a suitable product for us. Throughout the years recommendation systems became crucial in many different services, like Netflix, Amazon, Facebook, YouTube [21], sometimes becoming the most important part of the company. They help to keep customer attention or increase profits by accurately offering services or products. Even a small improvement of recommendation accuracy can yield high profit for a company. More accurate recommendations result in better user experience. They encourage users to spend more time watching films and listening to music, making the experience more addicting. In the case of shopping services, better recommendations increase sales. In recent years, the problem has been studied extensively, pushing boundaries of state-of-the-art solutions. Major scientific attention was reignited after the "Netflix Prize" competition. In 2006 Netflix started a contest in which participating teams were to improve upon the quality of Netflix's current prediction algorithms by at least 10%. This event begun a recommendation systems gold rush, the main prize being 1'000'000$ for the first team to beat the score. It took about three years until the winning solution appeared. The achievement was claimed by BellKor's Pragmatic Chaos team, and their system outperformed the initial algorithm by 10.6% [1, 13, 18, 24].

The general problem of recommender systems can be stated as follows: a set of users and a set of items are given. We will use the general term "items" - in practice they could be movies, songs, books, ads, shopping goods or even friends on social media sites. The goal is to build a model that predicts user preferences. Data used to train a model can vary between different approaches, which we can split into two distinct categories: content-based filtering and collaborative filtering. The former uses additional information about users and items to make predictions, while the latter is based solely on interactions between users and items. Those interactions can be divided into two subcategories, based on their type: explicit and implicit feedback.

1.1.1. Explicit feedback

As explicit feedback we classify all data that is given directly by the user, e.g. item ratings or written reviews. This type of feedback is priceless for services and companies. However, often only a small percent of users is interested in rating items and processing reviews requires additional approaches like sentiment analysis which increase a complexity of the recommender system.

1.1.2. Implicit feedback

Implicit feedback is collected without any interference with user interactions. Data gathered this way may still reflect user preferences; examples of such actions are watching videos, purchasing products or clicking ads. Unfortunately, it can be difficult to reconstruct user motives based on these actions.

1.1.3. Content-based filtering

Content-based filtering recommends items based on a comparison between the content of items and a user profile. For each item, we have to collect features like keywords in the title and description, tags, genre, director, etc. We also collect features of users like age, sex, preferences, previously bought products etc. Based on that, we define a similarity between users and items to recommend. This approach performs better than Collaborative filtering when we consider new users or items.

1.1.4. Collaborative filtering

In this case instead of assigning features to items and users, we only use a matrix containing information about interactions between users and items. This kind of data is much easier to gather: for explicit feedback we use user scores, for implicit feedback we assume all interactions are positive. The main issue with this approach is the lack of knowledge about new items. Every time we want to include a new item, we need to retrain the whole model. Similarly, items without interactions will typically be ignored by the system.

1.1.5. Collaborative filtering approaches

Matrix factorisation

The simplest approach is called matrix factorisation. Matrix factorisation algorithms work by approximating the user-item interaction matrix R with a product of two matrices, H and W. Each cell R_{ij} represents a value of user-item interaction when working on explicit feedback data or 1 when working on implicit feedback data. In case when we don't have information about interaction, we leave it empty - it doesn't take part in building the model. H contains latent factors of users, and W latent factors of items. We define the loss function as the difference between $H * W$ and R plus regularization [A]. We use gradient descent to optimize the loss function. Regularization and small latent sizes of H and W prevent the model from overfitting [A] . We pick items with best scores in $H * W$ to recommend.

Neural Collaborative Filtering

Introduced by X. He et al. [11], in this approach we use a neural network to predict scores, but the overall architecture is similar to a matrix factorisation model. In the latter, after

obtaining latent representations for a user and an item, we use their scalar product as the overall score. In NCF we use a nonlinear function, modeled by a neural network. This allows the system to learn complex interactions, and leads to increased performance.

Variational Autoencoders

Variational autoencoders for collaborative filtering were first introduced by D. Liang et al. [14] in 2018 as a state-of-the-art solution. We use their paper and accompanying code as a basis for our implementation, and provide a detailed description of this model in the next section. Currently, slight improvements have been proposed [25]. However, as our goal is to investigate computational improvements, we decided to use the original model.

1.1.6. Hybrid recommender systems

For completeness, we mention hybrid recommender systems that combine both content-based and collaborative filtering. This allows them to overcome difficulties related to both approaches. For example, a hybrid system could rely on content-based recommendations for new items, mitigating the cold start problem [A] in CF.

Chapter 2

Client requirements

A task presented by our client can be summarised into a few points:

1. Implementing neural network model according to the paper [14]. The neural network must have the same accuracy. This property can be validated by looking at metric results. They should be equal to those in paper. So our goal is to achieve the following values on MovieLens dataset:

Metric	Expected result
Recall@20	0.395
Recall@50	0.537
NDCG@100	0.426

2. Implementing downloading and parsing of 4 datasets: MovieLens [10], Netflix, Pinterest and Lastfm. It should convert the data into the format that can be fed to the model.
3. Creating training script. Script should iterate over previously parsed data, feed it to the neural network, calculate loss and improve wages. Also, it should support performance measurement and validating training progress.
4. Analyzing a possibility of optimizing training using Multi-GPU and Mixed Precision Training.

Chapter 3

Model

3.1. Theory

3.1.1. Variational autoencoder

An autoencoder [12] is a type of neural network designed to learn low-dimensional representations in an unsupervised manner [A]. Autoencoders consist of two parts: an encoder and a decoder. In our case, the encoder takes user interactions as an input and returns a latent representation, which contains fixed number of features. This representation is then forwarded to the decoder, which computes an enriched user interactions vector. 'Variational' means that an encoder returns two vectors:

- vector of means
- vector of variances

These are used to sample a randomized latent representation from a Gaussian distribution. This forces the network to store similar interaction representations nearby in the latent space, effectively smoothing it out. In the case of CF, this leads to improved performance.

The loss function used for training is a sum of three terms:

- negative log-likelihood
- KL divergence
- L2 regularization

Negative log-likelihood is equal to $-log(p)$, where p is the probability that picking randomly, from a multinomial distribution with probabilities proposed by the decoder, we get the users interaction history. We scale KL divergence loss by a parameter called annealing, which is increased during training, until it reaches a predetermined cap.

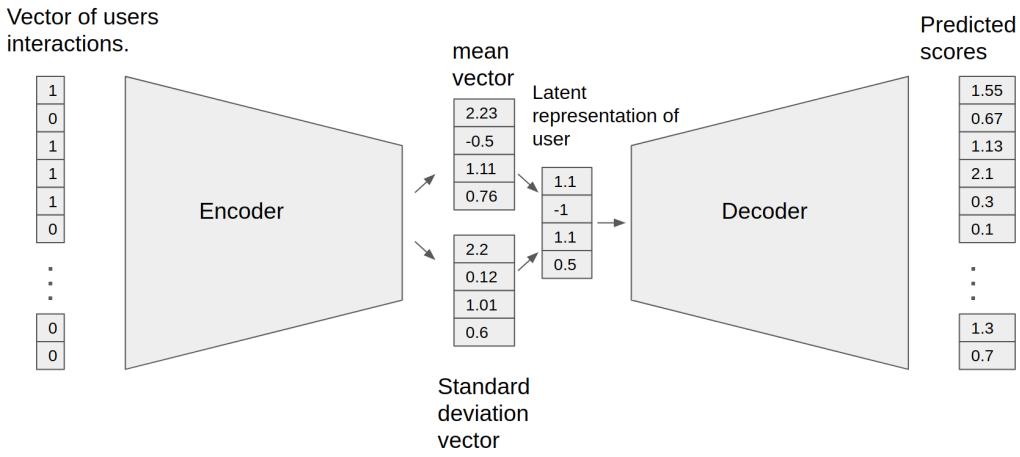

Figure 3.1: Autoencoder. Vector of user interactions contains zeros and ones, meaning interaction and no interaction respectively.

3.1.2. Metrics

To verify the accuracy of model's prediction, we adapt currently popular metrics used for evaluating recommendation systems, Recall and Normalized Discounted Cumulative Gain (NDCG). In general, for some new users, we feed the network with 80% of interactions and assess how well it recommended the remaining 20%.

Recall@K takes K best predictions, and measures the number of correctly predicted items (from the held-out 20% of interactions) divided by maximum number of possible hits.

$$Recall@K = \frac{|Relevant\_items\_to\_recommend|}{min(|Relevant\_items\_to\_recommend|, K)}$$

NDCG@K is a bit more sophisticated in comparison to recall. Recall ignores the order of predictions – the prediction with highest score has the same contribution as the item that barely got into the predicted set. Of course we would prefer the first prediction to be a hit, rather than the last. NDCG takes this into account by decreasing scores by:

$$DCG@K(u, \omega) := \Sigma_{r=1}^{K} \frac{I[\omega(r) \in I_u]}{log(r+1)}$$

where ω is a sorted vector of K recommendations, u represents a user, I_u is a set of relevant items and function $I[\omega(r) \in I_u]$ returns 1 if r-th recommended item is relevant.

$$IDCG@K(u, \omega) := \Sigma_{r=1}^{min(K,|I_u|)} \frac{1}{log(r+1)}$$

IDCG@K - Ideal Discounted Cumulative Gain@K yields a maximal possible value of DCG@K.

$$NDCG@K := \frac{DCG@K}{IDCG@K}$$

3.2. Implementation

In this section we describe optimization methods used to increase performance of our model. The aim of all modifications was to decrease training time while preserving scores on the following metrics achieved in the original paper: NDCG@100, Recall@20 and Recall@50. To implement the model, we have chosen the latest version of TensorFlow framework [7] - TensorFlow 1.13. TensorFlow is a leading open source platform for developing and deploying machine learning models. Although the core is written in C++, APIs for many popular languages exist. In our work, we use Python as it's most popular programming language in machine learning community.

3.2.1. Sparse matrices

The most effective and important optimization was the use of sparse matrices as an input to our neural network. Since it is very unlikely that any user would have interactions with most items, the input data is very sparse. For instance in the ML-20M dataset non-zero cells accounted only for about 0.5% of the whole matrix. Passing all these zeros explicitly to GPUs combined with short computation times leads to enormous bottlenecks. The use of sparse matrices dropped this derivable information and greatly increased efficiency of communication. However, as of TensorFlow 1.13, most operations on sparse matrices lack GPU support. For example, scalar products between dense and sparse vectors cannot be calculated. Although this necessitates ugly workarounds, our 12.5x speed-up during training justifies the means. Unfortunately, validation metrics have to be computed on CPU.

3.2.2. Data pipelining

To further decrease time spent on processing and loading data, we used TensorFlow Dataset API [19] for data pipelining. GPUs are able to greatly decrease the time needed to evaluate a single training batch, but for best efficiency we require a performant input pipeline that produces data for the next batch before the current has been processed. This reduces GPU waiting time to minimum and decreases epoch processing time almost twofold.

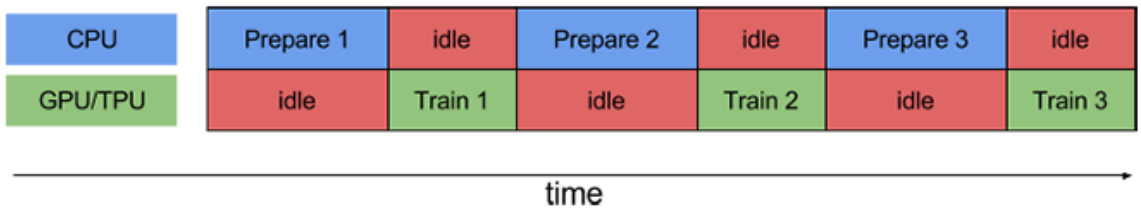

Figure 3.2: Before parallelization [2]

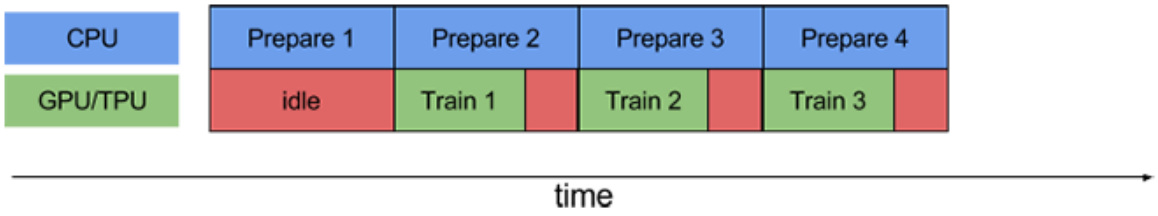

Figure 3.3: After parallelization [2]

3.2.3. Mixed precision training

Another idea to increase model's performance was to use 16 bit floats instead of 32 bit. This optimization's algorithm was presented and thoroughly described in *Mixed Precision Training* paper [15] written by Micikevicius et al. This approach resulted in great reduction of time of sending data between CPU and GPU. Taking less memory space input values allowed us to increase batch sizes which had positive effect on our metrics. However, having lesser floats results in lesser precision which was usually a problem when gradients' values were going below minimal non-zero FP16 value.

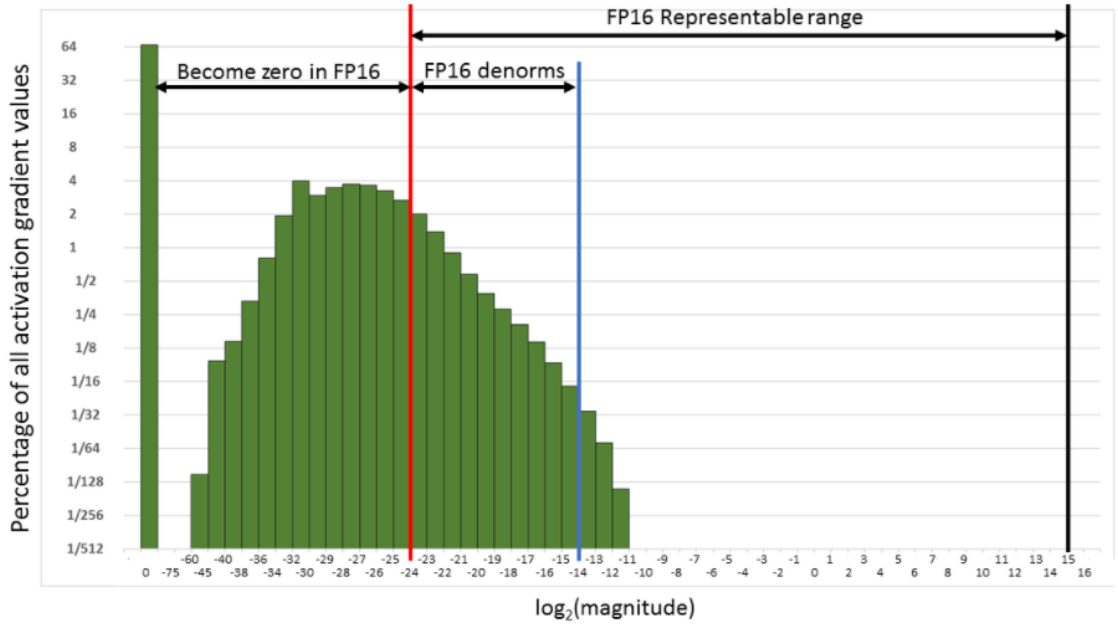

Figure 3.4: Histogram of gradient values [15]

The training was often diverging because in most cases the affected numbers were changed to zero without control. The solution of the problem is to scale them before gradient computation by multiplying values by some factor. It shifts them to the right to occupy range of FP16 values. Weight gradients must be unscaled before parameters update. In our implementation we used TensorFlow Automated Mixed Precision tool [16] by NVIDIA (we further use an AMP acronym). It is a feature of custom version of TensorFlow which is shipped with one of official Docker Images. It provides fully self-acting analysis of the graph and performs all mentioned above optimizations and necessary shifts. To use the AMP, we only have to import

it and turn it on. It's crucial that the dimensions of matrices are divisible by 8. Otherwise, tensor cores wouldn't boost the computations.

3.2.4. Multi-GPU

Data compression achieved by use of sparse matrices and mixed precision allowed us to effectively perform multi-GPU computations, since the communication was no longer an issue. Encouraged by good benchmarks and recommendation by NVIDIA we decided to introduce Horovod framework [22] into our implementation. Horovod is a distributed training framework for TensorFlow, but beside that it works with PyTorch [17] and MXNet [8] frameworks as well. It uses MPI [3] and NCCL [4] protocols in the background to achieve fast and functional concurrent platform. NCCL framework provides multi-node collective communication primitives that are optimized to achieve high bandwidth over NVLink high-speed interconnect used in DGX Station.

One of big benefits we noticed was ease of implementation of it in our code – the library provides wrapper for a neural network optimizer that automatically takes care of synchronization and managing connections. The overhead was low enough to get about 3.5 times speed-up with 4 GPUs.

Chapter 4

Usage

In this chapter we explain how to use our model.

4.1. Requirements

To run our model you need to meet the following requirements:

- Docker: Our model is distributed using docker image based on NVIDIA TensorFlow tensorflow:19.05-py3 image
- GPU with Tensor Cores: Tensor Cores are required to unlock a full potential of mixed precision training, but it isn't necessary. It could be trained even with CPU.
- Internet connection: Our repository doesn't contain datasets used to training and each of them has to be downloaded beforehand

There are 2 options of running the training.

4.2. Python script

4.2.1. Running instruction

First option is to run Python script called main.py. It's only possible to run script inside docker container, because AMP is only available this way. It isn't currently a part of any framework and isn't available in any package manager.

To build the docker image from Dockerfile [A], execute the following line:

```
docker build . -t nvidia_vae_running
```

Run the docker container

```
./scripts/docker/interactive.sh
```

Then you can run training with default settings and AMP

```
python main.py --train --use_tf_amp
```

4.2.2. Command line options

Script provides 3 basic run options

- train
- test
- benchmark

You should specify at least one from the above options. But you can use any subset of them. They are executed subsequently in the given order.

Train the model. The training is run at the beginning and might be omitted by loading pretrained model from export_dir.

```
--train
```

Test the model. The tests are run after the training, and print metric results. If the train option is not specified then the model is imported from directory export_dir.

```
--test
```

Benchmark the training. It prints average epoch training time and average epoch validation time.

```
--benchmark
```

Turn on Automatic Mixed Precision.

```
--use_tf_amp
```

Choose the dataset that the model is going to learn from. There are 4 possible datasets.

- `ml-20M`
- `netflix`
- `lastfm`
- `pinterest`

```
--dataset
```

In case of multiple GPUs available, you can specify its number.

```
--gpu_number
```

Number of GPUs used during multi-GPU training. If equals zero, VAE will use CPU.

```
--number_of_gpus
```

Number of epochs for training. The default value is 200 epochs and it's enough to reach the maximum accuracy (with default batch size).

```
--number_of_epochs
```

Batch size of training. The default value is 10000. It represents the number of users that is fed into the network at each step.

```
--batch_size_train
```

Batch size for validation and test. The default value is 10000.

```
--batch_size_validation
```

Frequency of validating the model. If it's set to n then validation is run once every n epochs.

```
--validation_step
```

Number of epochs to omit during benchmark. First epochs perform worse in comparing to next epochs due to framework initialisation. It is used only during benchmark.

```
--warm_up_epochs
```

Number of annealing steps

```
--total_anneal_steps
```

Annealing cap

```
--anneal_cap
```

Lambda λ – L2 regularization parameter[A]

```
--lam
```

Learning rate

```
--lr
```

4.3. Jupyter Notebook

The second option is to use Jupyter. The advantage is that you can create and view plots and charts easily.

Build the docker image.

```
docker build . -t nvidia_vae_running
```

Run the docker container, passing a port number for Jupyter Web Server.

```
./scripts/docker/interactive.sh <port_number>
```

Inside Docker container, run Jupyter Web Server.

```
./scripts/jupyter-run.sh
```

Open your browser, and paste the following URL.

```
http://localhost:<port_number>/notebooks/notebooks/VAE.ipynb
```

Click "Run" to run the training.

Chapter 5

Datasets

We wrote scripts that download and transform data from 4 different datasets:

Dataset	Description	Users	Items
MovieLens	Explicit user ratings of movies from movielens.org	138493	26744
Netflix	Explicit user rating of movies from Netflix. It was used during Netflix prize contest	480189	17770
Lastfm	Artists played by the user	358868	295041
Pinterest	Images pinned to boards. We treat boards as users and pins as items.	46000	2565241

We mainly focused on MovieLens dataset.

5.1. Transforming the data

Before working on dataset, it has to be cleaned. It contains users and items that have very few interactions with others, so modelling recommendations wouldn't make much sense – the same approach was in the VAE for CF paper [14]. This procedure is slightly different depending on the dataset. In general, the algorithm looks as follows:

1. If the feedback is explicit then we will cast it to implicit by treating rating above fixed threshold as 1 and otherwise 0.
2. Randomly shuffle the dataset
3. We remove items that didn't interact with enough users. Similarly, we remove users that haven't seen enough items. By default we set both thresholds to 5.
4. We split the data into train, validation and test sets. We had to get the following matrices:

training set	Contains 86 % of users. It is used in training the model.
validation_data_input	Contains 7 % of users and randomly chosen 80 % of their interactions.
validation_data_true	Contains 7 % of users and the remaining 20% of interactions.
test_data_input	Contains 7 % of users and randomly chosen 80 % of their interactions
test_data_true	Contains 7 % of users and the remaining 20% of interactions.

During validation and test we feed only 80% of interactions into the model. However we compare the output with all of them. This way we check if the model can properly predict the rest of the items.

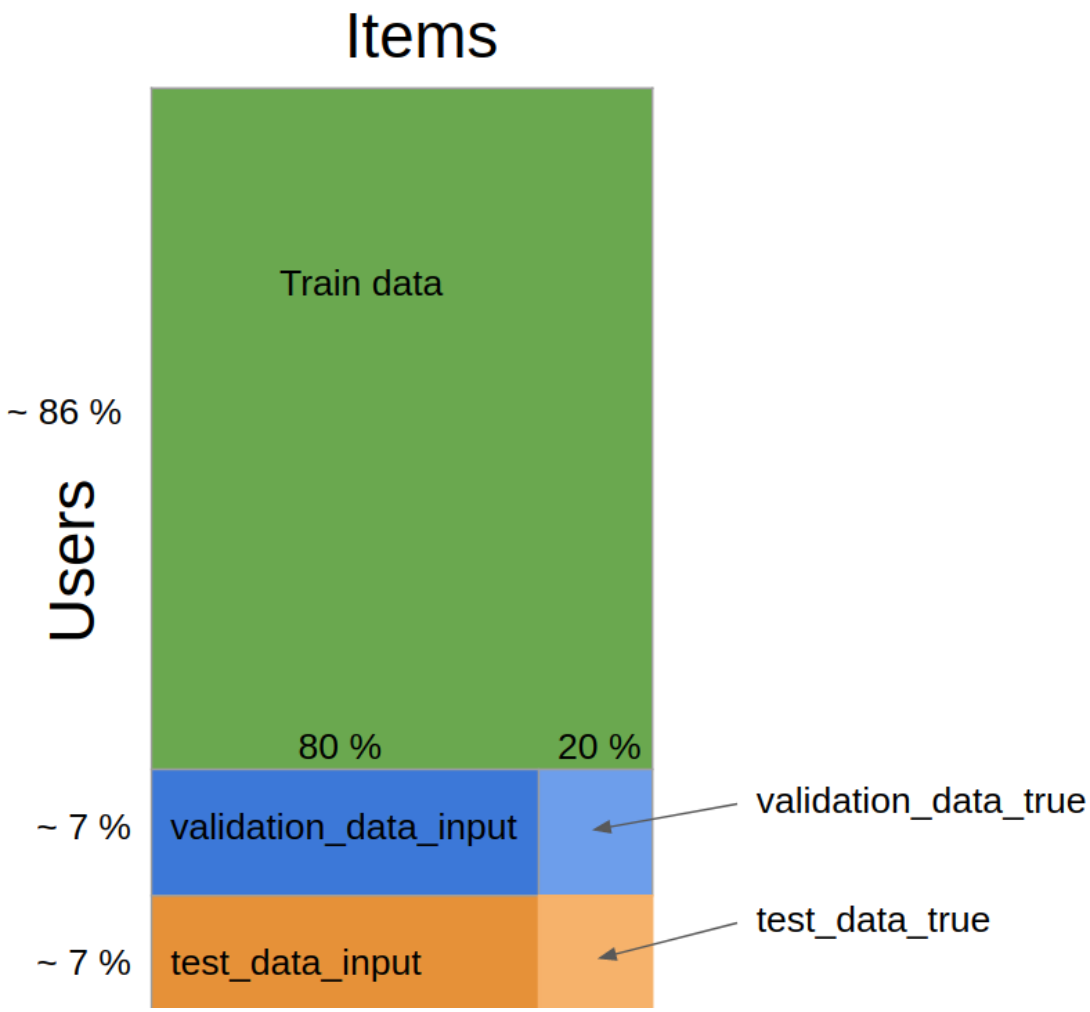

Figure 5.1: Divided ml-20m dataset

Chapter 6

Results

Having all optimizations applied we achieved total 145x performance boost on ml-20m with minor cost on metrics while preserving the same number of epochs and batch size. The most unclear result was gained on multi-GPU training because we had to adjust hyperparameters differently to fit the network to new training algorithm. We gained a lot out of tools provided by NVIDIA – automated mixed precision and CUDA processors did a lot of work and truly shortened training time.

6.1. Experimental platform

NVIDIA has given us an access to their DGX Station [5]. The DGX Station has following specification:

- 4 x Tesla V100 32GB
- Intel Xeon CPU E5-2698 v4 @ 2.20GHz
- 256 GB RDIMM DDR4 RAM

All of the experiments and measurements were conducted using this server.

6.2. Optimizations

In this section, we provide details about the results that we achieved after implementing consecutive optimizations.

6.2.1. Legacy model

We started from the legacy model that was attached to the paper [14]. As it hasn't been optimized for performance, one epoch took 23.25 s on average. The training batch size was fixed to 500 users. The plot below represents the time elapsed for every separate computation. As they are performed concurrently, they don't sum up to total epoch time.

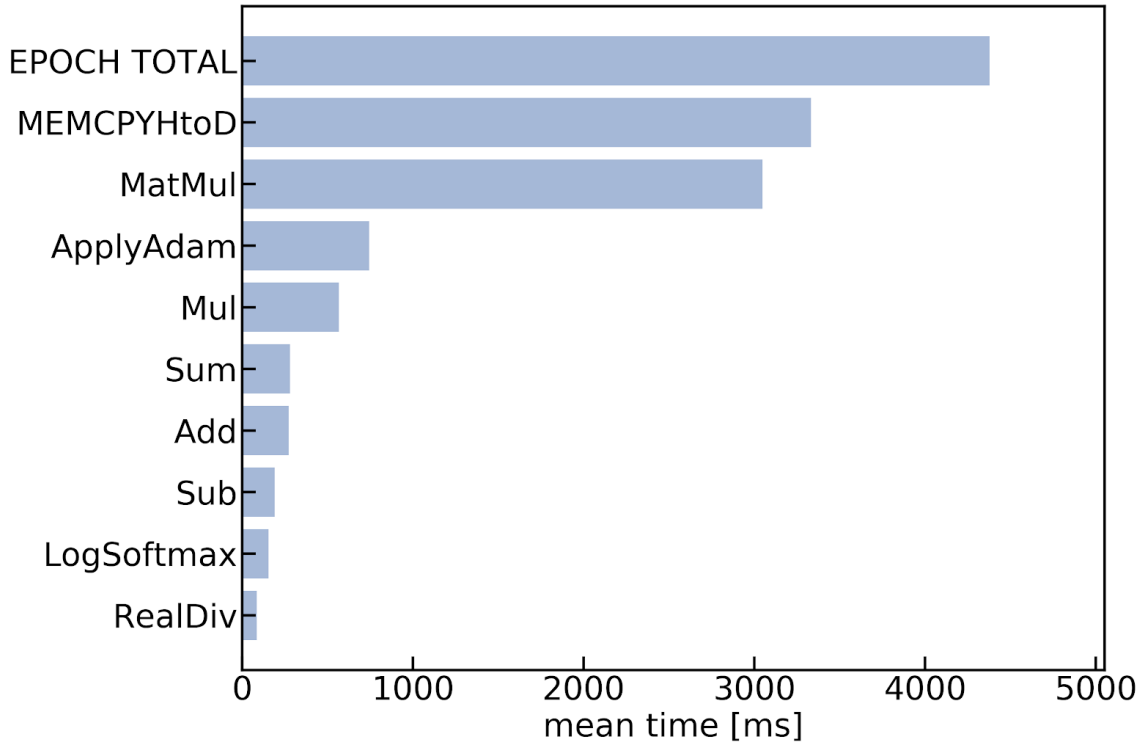

Figure 6.1: Combined operations time: legacy model

6.2.2. Sparse matrices

Sparse matrices structure requires explicit indexing of every non-zero element. This can be an issue if the number of them exceeds int32 range - which is apparently used in TensorFlow. Technically it was a huge limitation for us and blocked us from using full capabilities of our GPUs since we had 32GB of memory on our cards, but due to this issue, we could use only 17 of them this way. It improved performance significantly from 23.25 s per epoch to 1.86 s per epoch.

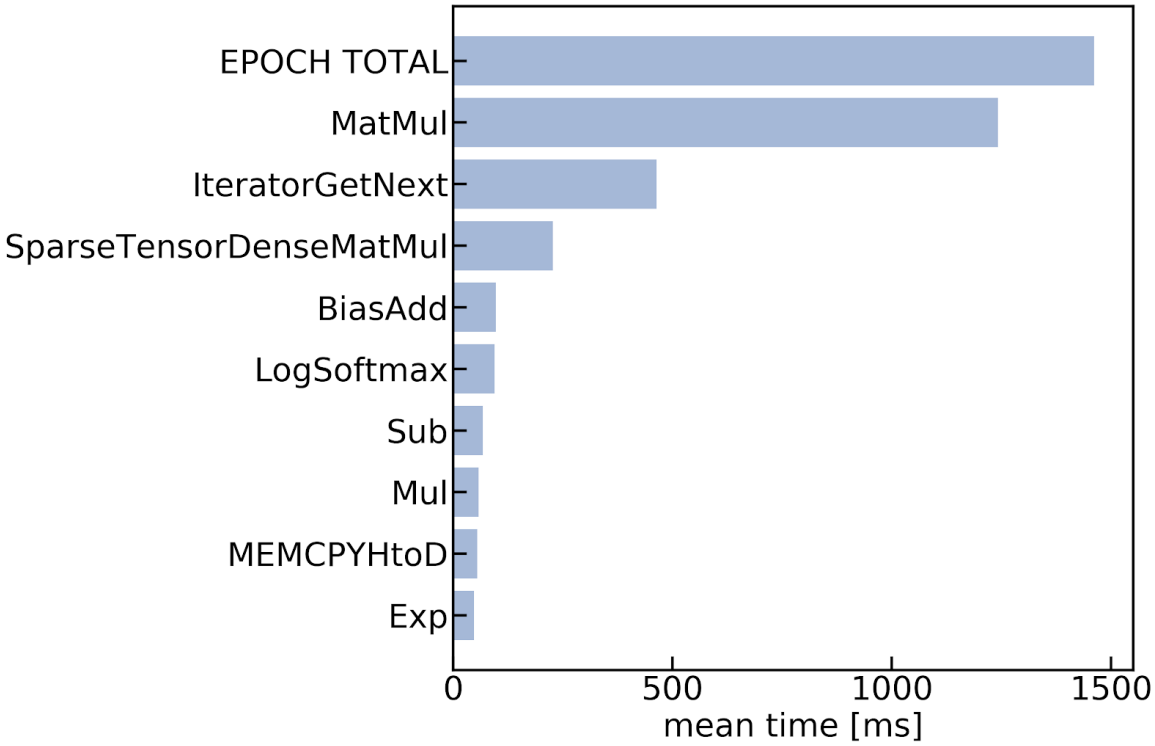

Figure 6.2: Combined operations time: sparse matrices

6.2.3. Data pipelining

Sparse matrices reduced the amount of data exchanged between CPU and GPU so much that data transmission takes less time than computations. Because we parallelized those 2 processes, the data transmission is running in the background and we are bottlenecked only by computations. It improved the overall performance from 1.86 s per epoch to 1.03 s per epoch on average.

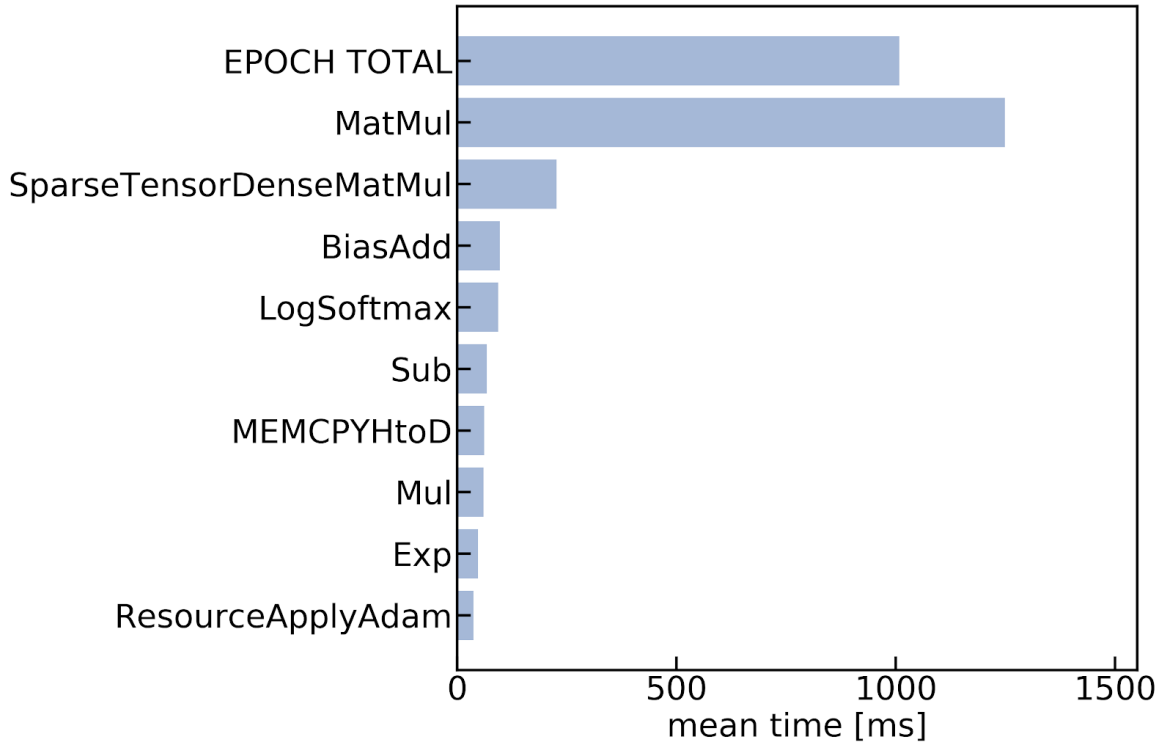

Figure 6.3: Combined operations time: data pipelining

6.2.4. AMP

Before AMP usage, the most demanding computation was dense matrix multiplication (MatMul on the plot). After enabling it, overall matrix multiplication time decreased by around 80% which is a really impressive achievement. Furthermore, the data exchanged between GPU memory and RAM has decreased. One of our disappointments was the fact that AMP doesn't currently provide speed up to operations on sparse matrices. It just leaves them without any changes to the graph. Summarizing, AMP improved the model's performance from 1.03 s to 0.56 s.

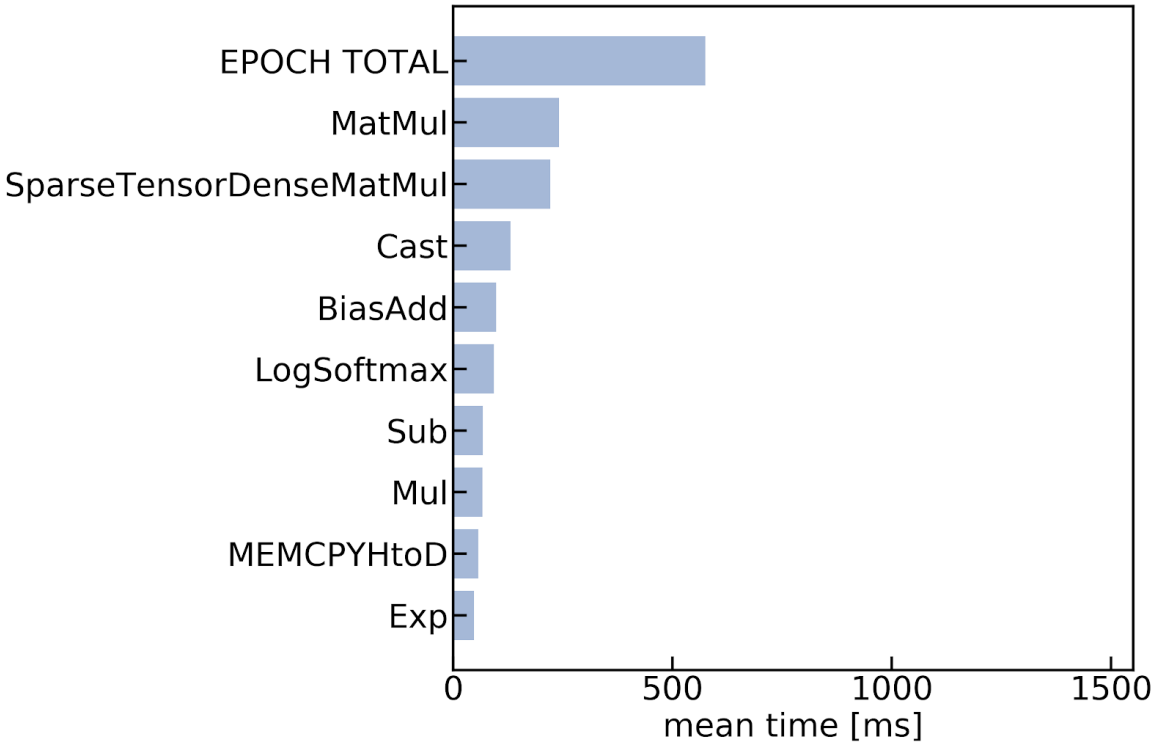

Figure 6.4: Combined operations time: AMP

6.2.5. Multi-GPU

Multiple GPUs greatly increased the performance of the model, offering about 3.5 times speed-up with 4 GPUs without a change on the metrics scores.

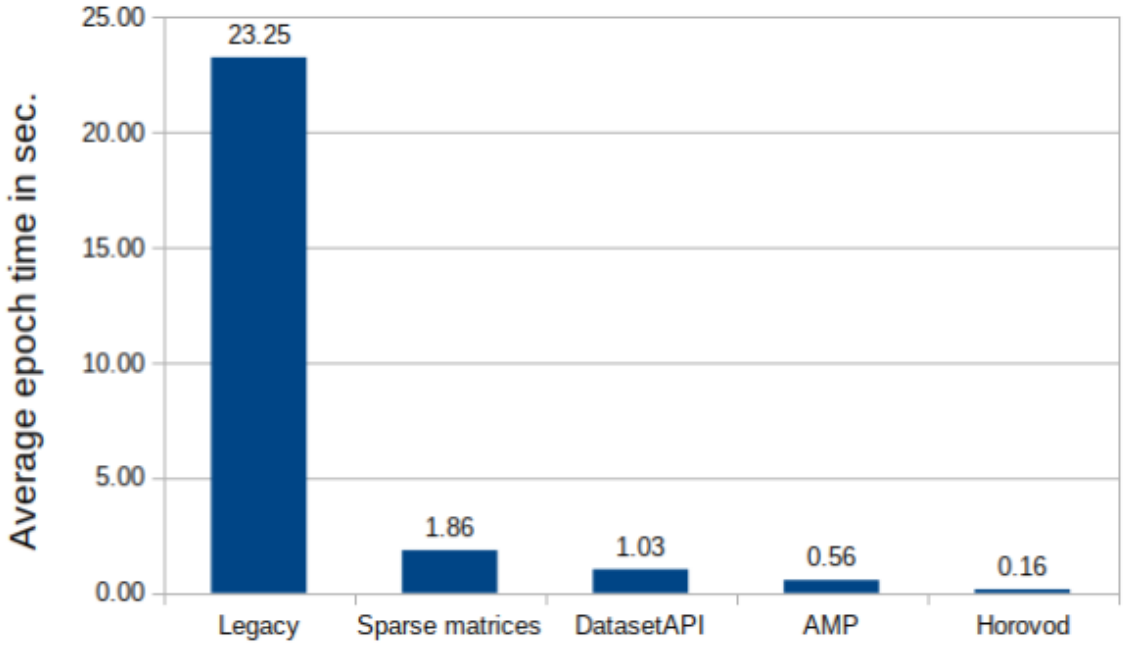

Figure 6.5: Epoch performance improvements

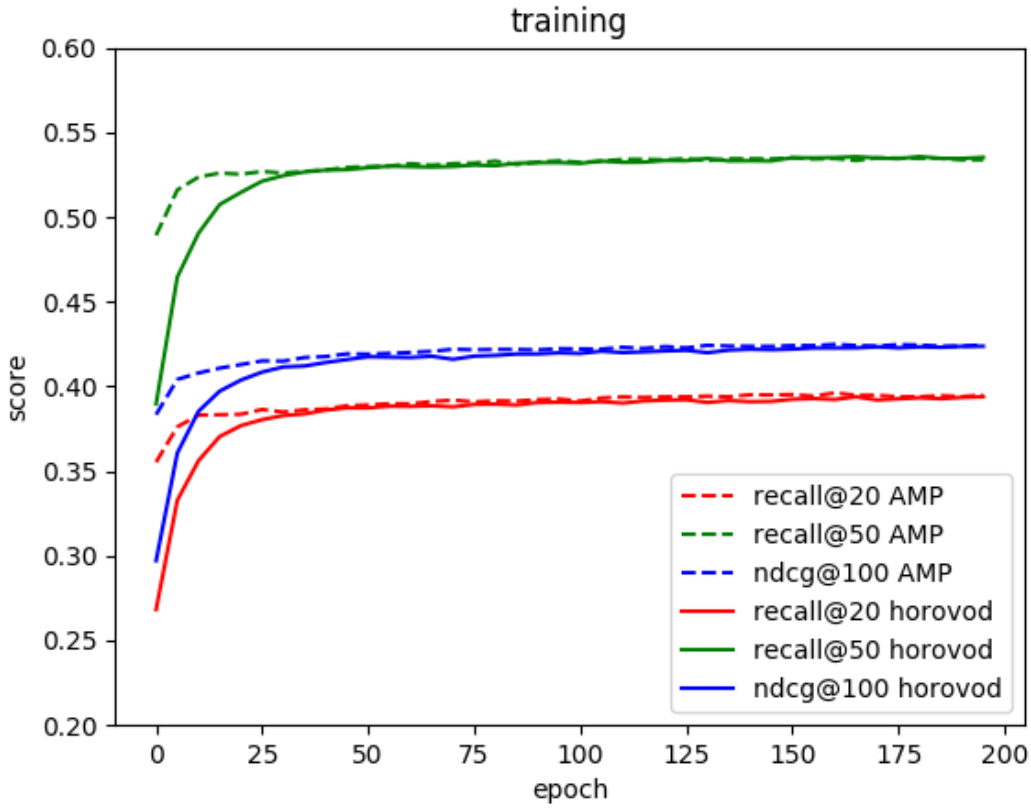

Figure 6.6: Comparison between metrics during training on single GPU in comparison with training on 4 GPUs using Horovod

6.3. Metrics performance

We have talked a lot about various speed improvements and their aspects. In this paragraph, I'd like to discuss how metrics results compare between a model presented in a paper with our optimized model.

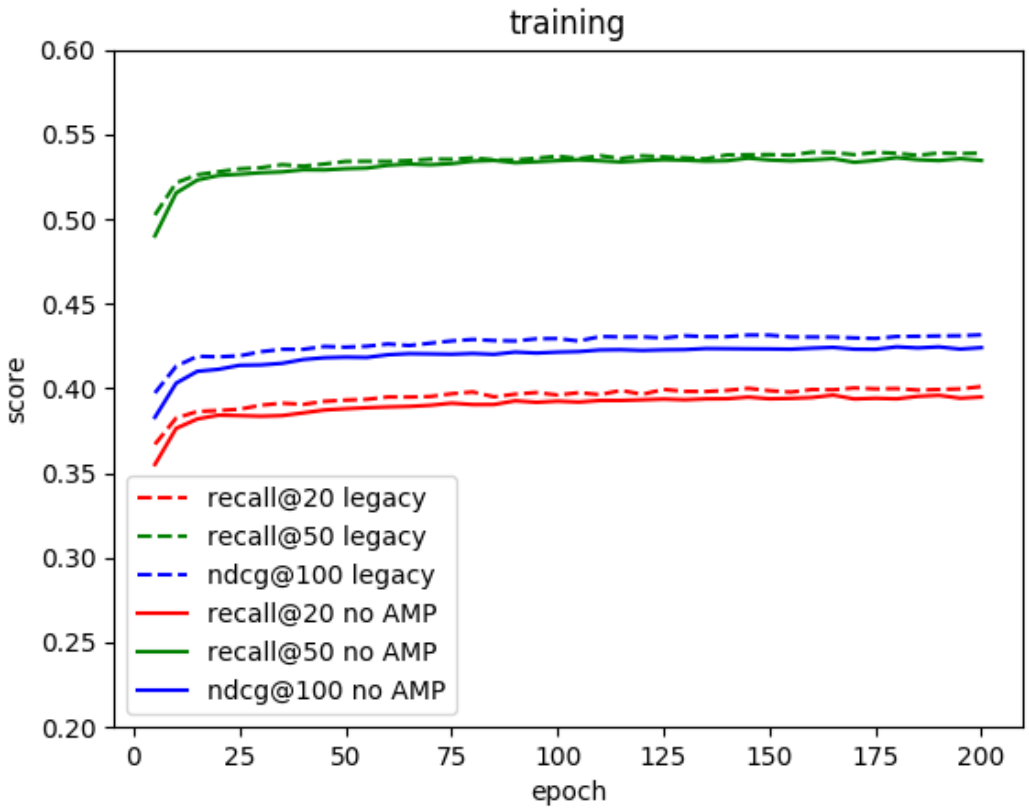

Figure 6.7: Comparison between metrics during training for implementation from paper and ours

As we clearly see, the optimized model underperforms a little. Why is that happening? Unfortunately, we can't give a clear, sure answer. We suspect that it can be related to two aspects:

- weights initialisation,
- usage of sparse matrix.

6.3.1. Weights initialisation

Trying to find a solution to the underperforming metrics, we started to simplify our architecture by removing some of its features like regularisation, sampling between encoder and decoder, dropout, sparse matrix usage etc. We ended with two models that differed only with computation graph declaration. Our model was implemented using `tf.keras.layers` package, while the original one used bare matrix multiplication and matrix addition. In theory, those models should be identical, but ours underperformed a little. However, when we changed the weights initialisation (Xavier initialization and Truncated Normal Initialization) [A] to setting all initial matrix values to zeros, the models started to perform identically.

6.3.2. Usage of sparse matrix

When we tried to observe similar behaviour on a model working on sparse matrices, its metrics results were once more a bit lower than the original model's metrics. Unfortunately, we cannot explain this behaviour. It may be related to some numerical issues or problems with a backpropagation on sparse matrices in TensorFlow, but it would require some further investigation.

6.3.3. Influence of mixed precision on metrics results

The last plot clearly shows that on our model, a mixed precision training has no negative effects on the process of training.

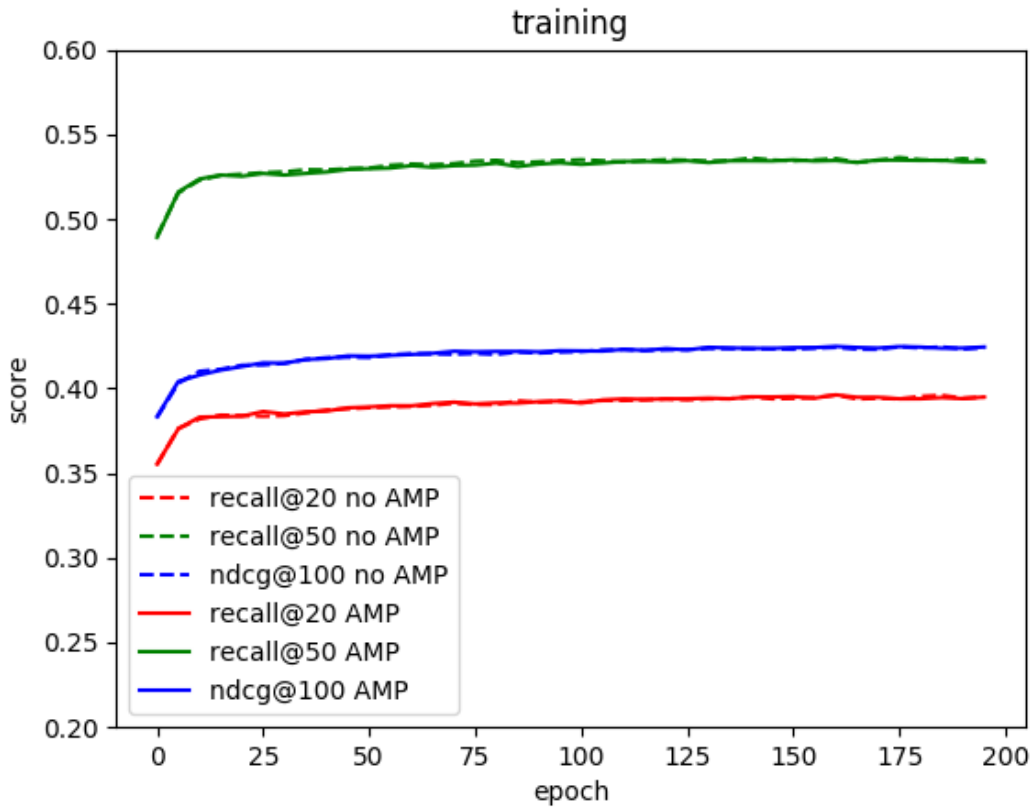

Figure 6.8: Comparison between metrics during training for model trained using FP32 and mixed precision

Chapter 7

Discussion

One of our most notable dilemmas was the choice between PyTorch and TensorFlow. After encouragement from the client side, we decided to develop our model using the latter. However, after encountering problems with sparse matrices support on GPU we started to wonder whether it was a good decision. TensorFlow style of model development by building a static graph of computations seems to be a great idea, but in our case with sparse matrices it required to build different placeholders for training, validating and querying the model. It is possible that those limitations are related to the architecture of GPUs, but we believe that PyTorch dynamic style could be really beneficial in terms of handling sparse matrices.

Next thing we would like to mention is support for automatic mixed precision training tools for both of the frameworks: Apex [6] for PyTorch and AMP for TensorFlow. Apex is a framework that has been released to the public in April 2018, while AMP was released just a few months ago with a few usage examples at `https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow`.

7.1. EstimatorAPI

EstimatorAPI [9] is a high-level TensorFlow API that greatly simplifies machine learning programming. It encapsulates the following actions: training, evaluation, prediction, saving the model. EstimatorAPI has been released in TensorFlow 1.1 in March 2017. EstimatorAPI has the following features:

- can be run on CPU, TPU or GPU without changing the model
- provides high level short, intuitive code
- builds graph for you
- provide a safe training loop that controls how and when to:
  - build the graph
  - initialize variables
  - load data using DatasetAPI
  - save the model

EstimatorAPI provides pre-made estimators like LinearClassifier or DNNEstimator, or enables to create customized ones. We used the latter option by passing our custom model. Additionally, we had to create proprietary datasets that provide training and validating data. Even though EstimatorAPI is a new, modern tool, it hasn't fit our needs, unfortunately, so we had to revert this idea. The problem is the training function is slow and it's not easy to optimize using our methods. Writing our custom metrics for the model was also extremely difficult and problematic.

7.2. Calculating metrics on GPU

Validation data contains more than 10 times less users than training data. However, initially calculating all necessary metrics took 10 times longer than learning through the whole epoch. It involves moving the output from GPUs to CPU and performing metrics calculations. Keeping in mind issues with sending data to and from GPU in the first implementation and knowing GPU capabilities to outperform CPU in floating-point operations, writing metrics in TensorFlow on GPU seemed to be a great idea. Unfortunately, TensorFlow has very scarce support for aggregating data in matrices on GPU, both for sparse and dense matrices. Information about this is extremely elusive – we found no website, blogpost nor any piece of documentation that would warn about possible issues. We have spent a lot of time trying to manoeuvre between those framework restrictions and limitations, but with no positive results.

During our fights with sparse matrices on GPU, we considered implementing missing functions on our own using CUDA. This could eliminate our toughest issues and would open new ways of optimization, but it would be definitely the hardest challenge to do. Finally we settled that it would require too much time to learn CUDA and get into TensorFlow's code to get this done. Also distributing our solution with a customized framework would be much more difficult than just posting some code on GitHub – we don't expect that people would be interested in compiling and building a whole TensorFlow from scratch.

7.3. How metrics scores relates to reality

We conducted the following experiment: We marked a few movies from a specific film series and fed it to our implemented VAE. We were expecting that the model would return other movies from the same series. We tried with James Bond and Star Trek series. What was the outcome? Actually, the model hasn't recommended any movies from those series. Why? The reason is that the people don't restrain themself to watch only one series. Many of us watch also different genres or series. The model is as good as the data that was used for training. This experiment showcases that our expectations for recommendation may be surprisingly weakly correlated with model's recommendation.

Chapter 8

Conclusion

Within 10 months of working on the model, our four-person team managed to speed-up the training almost 150 times with only a little loss of accuracy. To achieve this result, we had to deal with several bottlenecks and use some hardware-based optimization techniques to our advantage. Some of them, like AMP or Horovod, may at first look like they are almost out-of-the-box solutions. However, reality can be a bit harsh and painful – the technology that we were using was very recently released, so it's still not well documented and there are no online communities using them, who could help or advice. Fortunately, we managed to cope with those difficulties.

In the beginning, we found out that efficient data transfer is crucial for the project success because sending dense matrices and multiplying on the first layer takes a significant amount of training time compared to the rest of computations. It was essential to improve transmission to be able to count on getting any gains from the use of multiple GPUs. Without this optimization, other methods wouldn't help much as they could speed-up only computations. When we started our project, we haven't expected that our task will mainly focus on how to effectively work with data. We spend a lot of energy changing dense matrices into sparse. We had to deal with limited sparse operations available on GPU and we came across many dead ends. We believe that TensorFlow should provide more GPU kernels for basic operations on sparse tensors or at least inform users explicitly in documentations about possible issues.

The last steps were the features that were most important for our client – Automatic Mixed Precision and multi-GPU using Horovod. Although, they are supposed to be simple to use tools, their integration took quite long time because of our lack of experience. What is more important are their effects on model's speed-up – our main goal of the project, which turned out to be quite impressive: halved time using AMP and around 3.5x speed-up using 4 GPU.

In the end, we and our client are quite proud and satisfied with the outcome of the project. We proved that mixed precision training and multi-GPU can really reduce training time even relatively shallow model like variational autoencoder for collaborative filtering.

Chapter 9

Team members contribution

9.1. Albert's contribution

- Implementation of:
  - Data loading and preprocessing
  - TF-AMP usage
  - FP16 precision research
  - Data pipelining

9.2. Michał's contribution

- Organising the team's work, coordinating contacts with NVIDIA representative
- Implementation of:
  - Metrics
  - Scripts for end users

9.3. Frederic's contribution

- Implementation of:
  - Neural network architecture
  - Multi GPU training
  - Metrics

9.4. Radosław's contribution

- Working environment setup
- Implementation of:
  - Multi GPU training
  - TF-AMP usage

Chapter 10

CD contents

The CD contains the code of our project. The root directory contains the main script and Docker files. There are following subdirectories:

- notebooks – Jupyter Notebooks
- vae - data loading, model definition and training script
- scripts - bash scripts that runs Docker and Jupyter

Chapter 11

Acknowledgement

We would like to thank NVIDIA Corporation for providing us with professional hardware without which performing necessary tests would be impossible, with special mention of Tomasz Grel for his assistance and support during creation of this project.

Bibliography

[1] https://www.netflixprize.com/leaderboard.html.

[2] https://www.tensorflow.org/guide/performance/datasets.

[3] https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[4] https://developer.nvidia.com/nccl.

[5] https://www.nvidia.com/en-us/data-center/dgx-station/.

[6] https://devblogs.nvidia.com/apex-pytorch-easy-mixed-precision-training/.

[7] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[9] Heng-Tze Cheng, Zakaria Haque, Lichan Hong, Mustafa Ispir, Clemens Mewald, Illia Polosukhin, Georgios Roumpos, D. Sculley, Jamie Smith, David Soergel, Yuan Tang, Philipp Tucker, Martin Wicke, Cassandra Xia, and Jianwei Xie. TensorFlow Estimators: Managing Simplicity vs. Flexibility in High-Level Machine Learning Frameworks. *CoRR*, abs/1708.02637, 2017.

[10] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, December 2015.

[11] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. *CoRR*, abs/1708.05031, 2017.

[12] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013. cite arxiv:1312.6114.

[13] Yehuda Koren. The BellKor Solution to the Netflix Grand Prize. https://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf, 2009.

[14] Dawen Liang, Rahul G. Krishnan, Matthew D. Hoffman, and Tony Jebara. Variational autoencoders for collaborative filtering. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 689–698, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

[15] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *CoRR*, abs/1710.03740, 2017.

[16] NVIDIA Corp., https://docs.nvidia.com/deeplearning/frameworks/tensorflow-user-guide/index.html#tfamp. *TensorFlow Automatic Mixed Precision*, 2019.

[17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[18] Martin Piotte and Martin Chabbert. The Pragmatic Theory solution to the Netflix Grand Prize. `https://www.netflixprize.com/assets/GrandPrize2009_BPC_PragmaticTheory.pdf`, 2009.

[19] Etienne Pot, Afroz Mohiuddin, Pierre Ruyssen, Marcin Michalski, and Ryan Sepassi. Introducing TensorFlow Datasets. `https://medium.com/tensorflow/introducing-tensorflow-datasets-c7f01f7e19f3`, 2019.

[20] raffi. New tweets per second record, and how! `https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html`, 2013.

[21] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor. *Recommender Systems Handbook*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.

[22] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[23] Susan Wojcicki. Industry Keynote with YouTube CEO Susan Wojcicki. `https://youtu.be/O6JPxCBlBh8?t=645`, 2015.

[24] Andreas Töscher and Michael Jahrer. The BigChaos Solution to the Netflix Grand Prize. `https://www.netflixprize.com/assets/GrandPrize2009_BPC_BigChaos.pdf`, 2009.

[25] Jingbin Zhong and Xiaofeng Zhang. Wasserstein autoencoders for collaborative filtering. *CoRR*, abs/1809.05662, 2018.

Appendices

Appendix A

L2 Regularization – A function of neural network parameters, that is added to the loss. It is equal to the root of the sum of squared parameters.

KL Divergence – A function of 2 probability distributions. It returns how different they are. In our case we compare our distribution with Gaussian.

Unsupervised learning – Learning without labeled data.

Annealing – During training we increase the importance of KL-Divergence in loss by multiplying it by an anneal factor. We stop when an anneal cap is reached.

Overfitting – The situation when a Neural Network is closely fitted to the training set that it is difficult to generalize and make predictions for new data.

Regularization – Adding terms to loss, that prevent then Neural Network from overfitting.

Xavier initialization – Initializes the parameters of Neural Network. Tries to preserve the scale of gradients.

Truncated Normal Initialization – Initializes the parameters according to Truncated Normal Distribution.

Dockerfile – The file that lets you build a "virtual" operating system.

Cold start problem – In collaborative filtering we don't know anything about new users, because they didn't see any items. So, we can't give them any personalized recommendations.