FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Merlin: Recommender Systems pipeline implemented on GPUs
Student:	Čeněk Sůva
Supervisor:	Ing. Petr Kasalický
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2022/2023

Instructions

Explore the area of Recommendation Systems and the NVIDIA Merlin™ open-source framework. Describe the basic concept of the Merlin™ framework and compare it with other standard technologies in Python. Examine the approaches used in Recommendation Systems that synergize with the Merlin framework, choose one and implement it. Select suitable non-trivial input data and perform an experiment to measure throughput and performance (e.g. recall) of the implemented approach. Discuss the benefits and limitations of your solution in terms of computing time, memory, scalability and hardware needed.

Electronically approved by Ing. Karel Klouda, Ph.D. on 21 January 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Merlin: Recommender Systems pipeline implemented on GPUs

Čeněk Sůva

Department of applied mathematics
Supervisor: Ing. Petr Kasalický

May 12, 2022

Acknowledgements

First and foremost I would like to thank my supervisor Ing. Petr Kasalický for his guidance, motivation, and his time provided when writing this thesis. I would also like to thank him for helping me in creating a thesis that exists for more than just a title. I would like to thank my girlfriend for cooking me meals and taking care of me. Next, I thank Recombee, namely doc. Ing. Pavel Kordík Ph.D., for providing an environment where I could work with people who have similar interests to mine.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 12, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Sůva, Čeněk. *Merlin: Recommender Systems pipeline implemented on GPUs.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Tato práce do detailu popisuje aplikační framework NVIDIA Merlin™ a všechny jeho části. Dále popisuje jak Merlin využít při implementaci škálovatelného předzpracování dat, návrhu a implementaci modelu, a generování predikcí v rámci doporučovacího systému. Merlin je společností NVIDIA prezentován jako jednoduchý pro použití a vysoce škálovatelný framework schopný zvládnout velikost dnešních dat. Tato práce ověřuje výroky společnosti NVIDIA a hodnotí Merlin z pohledu vývojáře. Dále popisujeme obecné přístupy v doporučování a optimalizace implementované společností NVIDIA. Poté popíšeme, jak využívat framework Merlin a jeho moduly. V prostředí Merlin se nám podařilo naimplementovat doporučovací systém, jehož výstupem jsou relevantní výsledky.

Klíčová slova doporučovací systém, využití GPU, aplikační framework Merlin™, předzpracování dat, hluboké učení, inference

Abstract

This work describes the NVIDIA Merlin™ application framework and all its parts in detail and how it can be used when implementing scalable data preprocessing pipeline, designing a model architecture, training a model, or generating predictions for recommendation system. Merlin is presented by NVIDIA as easy-to-use and highly scalable framework capable of handling real-world data and workloads. This thesis verifies claims made by NVIDIA and reviews Merlin from developers perspective. We describe the general approaches used in recommendation systems and then discuss the optimizations NVIDIA implemented. Next we provide a guide in how to use some of the Merlin modules. We have managed to use Merlin components to implement a movie recommendation system that is producing relevant results.

Keywords recommendation system, GPU utilization, Merlin™ application framework, data preprocessing, deep learning, inference

Contents

List of Figures

Introduction

Motivation

Large amount of content is uploaded to the Internet every day. This content ranges from anything digital, such as videos, pictures, and news posts, to actual physical items that a customer might buy from an e-Shop. To customize the browsing experience, many services leverage some kind of recommendation system. This increased need for unique experience creates a demand on the people creating these systems, which in turn creates interest in automation and simplification of the process.

There are many tools available to help with the implementation of a recommendation system. There are also many people who are just beginning to explore the options available to them. The decision to use a tool for a job can be critical to the process. This thesis addresses this problem by analyzing available frameworks and conducting an experiment with one of them.

We will go through what a recommendation system implementation and usage looks like. Then, we compare commonly used frameworks, pointing out the major differences when compared to the one we use. We have selected to experiment with the open-source application framework developed by NVIDIA called Merlin™. We will implement a pipeline consisting of data pre-processing, creating and training a machine learning model, and model inference. The benefits and drawbacks of Merlin™ are then evaluated based on the tool as a whole, including documentation, simplicity of use, performance impact and versatility.

Goals

The goal of this thesis is to describe how a recommendation system works, how it is built and used. After this introduction to the domain, we will analyze Merlin™ and compare it to well-known Python frameworks like Pandas. Combining this knowledge, we design a recommendation system pipeline. Choosing

the data set is an important part of the process, as we want to leverage the capabilities of the framework. Preferably, we choose an artificial neural network architecture that is supported in our framework. Once this is done, we evaluate our model in terms of precision and performance.

Chapter 1

Recommending Introduction

1.1 Recommendation System

Recommendations systems are used today by most of the services that provide more products and items that a user can browse through. In this chapter, we analyze the basics and some of the more advanced techniques used in reccomending. We also analyze tools for building recommendation systems, with a special focus on NVIDIA Merlin™, an open source recommendation system framework.

1.1.1 Overview

Recommendation system is a system that encapsulates a method to recommend any arbitrary item. Recommendations are generally served and tailored to the needs of each user. When the system cannot identify the user, it can resort to recommending items popular amongst large clusters of users. When the system can identify the user and has some historical data stored about them, it can serve items more relevant to each and every user.

Recommendations are vital to the user's browsing experience. The amount of content available today is well beyond the limit that a single human being can browse through, let alone inspect. Machines can help us with this problem as long as we are able to formulate the problem mathematically.

The need for such systems exists when an application begins to provide large amounts of content, as is not uncommon these days. There are many examples for this. Let us say that we want to watch a video; what site do we choose? Whatever the answer, this site will most likely provide videos covering multiple topics, such as gardening, cooking, sports, architecture, programming, and more. When a user engages positively with any such topic, we can say that the user finds it interesting and would like to see more of it.

The well-known approaches are content-based filtering and collaborative filtering. When content-based and collaborative filtering is combined, it is

called a hybrid approach. The fourth option would be to filter based on demographics. Content-based filtering focuses on the properties of each item individually. We can find similar items on the basis of these properties and recommend them. In collaborative filtering, we focus on finding a similarity between users. Then, when we know that a user engaged positively with an item, we find a similar user and recommend the same item to him/her. [8, chapter 6]

The goal here is to individually predict for each user the items with which this user would like to interact. User-item interactions that did not yet happen can be assigned a value, a prediction. When we rank these items for each user by their predicted value, we can then select the top few, or use a different approach, for the items we will recommend.

1.1.2 Data

A data set usually contains users, items, and interactions. This information can be obtained implicitly or explicitly. Implicit information is any information recorded by the system. Some examples of implicit information are view of an item, time spent watching a single video, or time spent on the site on average per day. Explicit information is any information provided by the user on purpose. Explicit information can be a like or a dislike, product rating, comment, or any kind of reaction that the user explicitly provides when asked.

Information comes in various forms. It can be categorical or numerical. We further split categorical information into nominal and ordinal information, depending on whether an order can be found in it. Education usually can be sorted because elementary school comes first, then high school, and so on. In color, however, no such phenomena can be found.

Users can have multiple attributes, like age, address, or sex. Some even more specific would be information on whether the user has kids, education, color preference, etc. Items can have color, genre, duration, size, and so on. Interactions usually have some kind of value. This is useful when we want to distinguish between a movie rating of 1 star out of 5 and 3 stars out of 5. Timestamp or some other type of time information is also important because the user's preferences change in time. A typical example of this are seasonal items, such as skis and swimsuits.

Interactions are often stored in an interaction matrix. The rows of the matrix are individual users and the columns correspond to items. The interaction matrix is usually sparse as users cannot possibly interact with more than a fraction of the items. An example of interaction matrix with M users and N items containing item rating from 0 to 5 is illustrated in Figure 1.1.

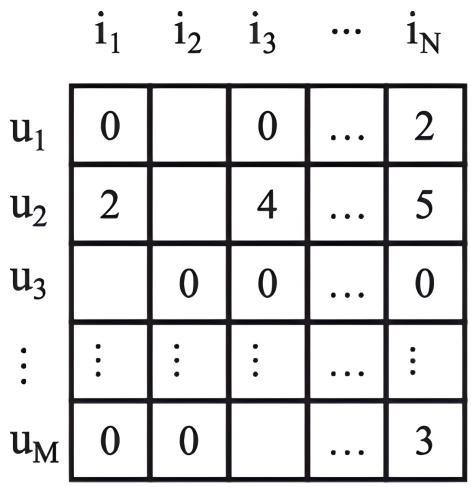

Figure 1.1: Interaction matrix

1.1.3 Machine learning

Machine learning is currently the basis for successful large-scale recommendation systems. It refers to learning a model, for whatever task at hand, by leveraging the information stored in our data. In our case, we will predict the value of an interaction.

Interaction data is a real-world example of what a user prefers, or which two items go well together. When learning our model, we examine the outputs it gives us for already known interaction, that is present in our data set. When the model's prediction is off, we have to adjust it's prediction process so that it will give more accurate prediction the next time. We repeat this process until we are satisfied with the model results. The process of retrieving a prediction from a machine learning model is called inference.

Predictions can be made beforehand and saved. In large-scale systems however, the prediction is not being calculated for every user-item pair, but rather after retrieving some more-likely candidates[wnd paper or specific retrieval literature]. Generation of predictions costs time and energy. Retrieval is a process where we retrieve a few hundreds to thousands candidate items using more traditional queries based on logical operators, similar to SQL. Only for items retrieved are user interactions predicted and ranked.

1.2 Pipeline

Machine learning pipeline in our context defines the following steps:

1. Transformations applied to raw input data.
2. Definition of a machine learning model.

3. Model training

4. Model inference

The pipeline should be repeatable to allow us to execute a fast prototype when we introduce some changes. Prototyping is what makes a workflow efficient and flexible. Obtaining the data set (e.g. download from a website) is not part of the pipeline, as this usually needs to be done only once, as long as we do not overwrite the original files.

Information from websites and other systems is usually stored in a format relevant for, and readable by humans, such as strings. Computers at their core always work with numbers and this makes data preprocessing a necessary part of our pipeline. Enhancing the data with features extracted from already existing ones can also be a process that significantly improves predictions. Extracting a season, day of the week, and time of day provides a critical information from a simple feature like timestamp. This process is called preprocessing and feature engineering. In Merlin, the part responsible for this part of the pipeline is NVTabular, discussed in Section 2.3.

Model definition will be included in the pipeline, as the structure changes when we add new features, or change the approach to processing a feature in a way not compatible with current state of the model. In the experimental process, this will happen more often than not. Finding the best model architecture is usually the result of extensive experimentation that takes a lot of time. Experimentation is the core reason we try to optimize testing an architecture. When the model is compiled, we need to train it on the data set. The following step after training is to save the model to disk, so it is stored and our training progress cannot be lost by simply restarting the machine or Python kernel in our case.

After the model has been saved, it will be served using an inference server. In our case, we inject the model into a Docker container. The container then runs a server, and we make requests to infer predictions. Besides metrics like accuracy and recall that are evaluated on the whole data set, we also would like to see specific item recommendations for specific users. This is a part of our pipeline, because we want to see for ourselves whether the predictions could be relevant for the user or not. An example of this could be a user purchasing only jeans from an e-shop multiple times. Such a users recommendations should certainly contain jeans, as they clearly are to the customer's liking.

Chapter 2

Analysis

2.1 Merlin™

Merlin is an open source application framework and a library collection that aims to ease the process of building a high-performance recommendation system. Merlin provides tools to help with implementation of all the parts of a pipeline discussed in Section 1.2. The architecture is shown in Figure 2.1. Consisting of multiple components, Merlin allows parts of the pipeline to be substituted with other standard technologies, like TensorFlow, to create and train an artificial neural network.

Merlin is currently in development and often undergoes changes. New features and components are added regularly. The developers addressed the

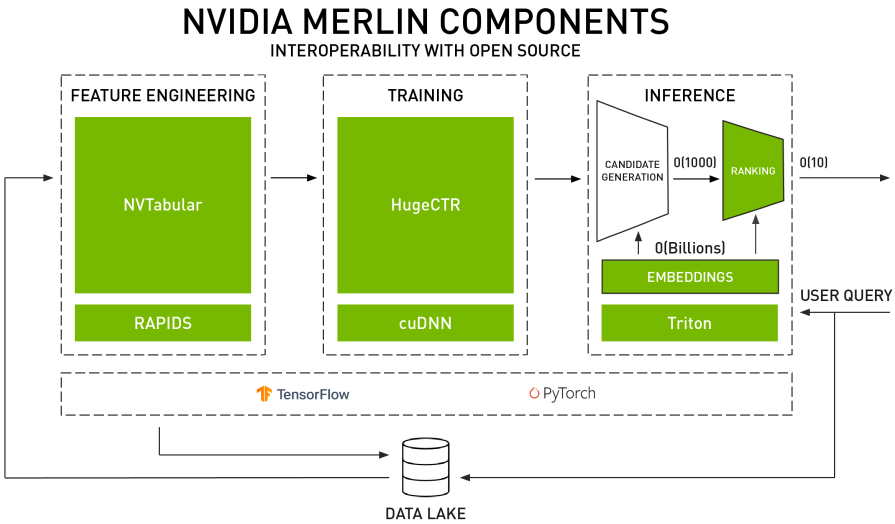

Figure 2.1: Merlin architecture[1]

changing nature of their product by containerization. Docker containers, properly versioned and customized depending on use are available from NVIDIA's catalog site[9]. Merlin builds on some other available libraries that will be presented and described shortly, like cuDF, Dask, and also uses Triton Inference Server as inference mechanism.

2.2 cuDF

CuDF is a library that allows us to load, manipulate, and save data on GPU, rather than CPU, as is typical. CuDF offers an API that mirrors Pandas' API. The difference, important for us, is that it utilizes a high memory bandwidth and processing power of the GPU, on more of this later in Section 2.3.4. GPU utilization is the reason why cuDF was created. It tries to provide the same API as Pandas, because Pandas is one of the most commonly used libraries for tabular data operations. One of the prerequisites for this is CUDA installed on the system.

Dask is a cuDF superstructure that supports workloads we would like to distribute across multiple GPUs. This comes handy when a data set does not fit a single GPU memory or when we want to utilize multiple GPUs processing power. Dask handles the logic behind splitting the dataset into partitions and distributing them to specific GPUs. It provides a high-level API for those who do not wish to delve deep into CUDA programming.

2.3 NVTabular

NVTabular is one of many Merlin components. In the pipeline, it is responsible for feature engineering and other forms of data set manipulation. It is built on cuDF and Dask (see Section 2.2). NVTabular provides a high-level API for manipulating the data, allowing us to define what we want to do with the data instead of what and how. In this sense, NVTabular tries to act similarly to functional programming languages. As the name suggests, NVTabular is suitable for handling tabular data, which are organized into rows and columns. If we would like to include images (or other unstructured data) in our pipeline, we shall look for a different tool for preprocessing. According to the documentation[10] it can handle data up to terabytes in size.

Workflow and Operator classes are the building blocks of the framework. Workflow is a graph (directed acyclic graph) of the operations that will be applied to the data set once it is loaded in memory. Workflow consists of Workflow Nodes (e.g. Figure 2.2), that are created by applying Operators to another Workflow Node, or Column Selectors (described in Section 2.3.2). Operators can be applied to a list of columns all at once. There are two types of operators: *Operator* and *StatOperator*. Operator performs such actions that do not need any calculated values from the data as a whole to be applied. In

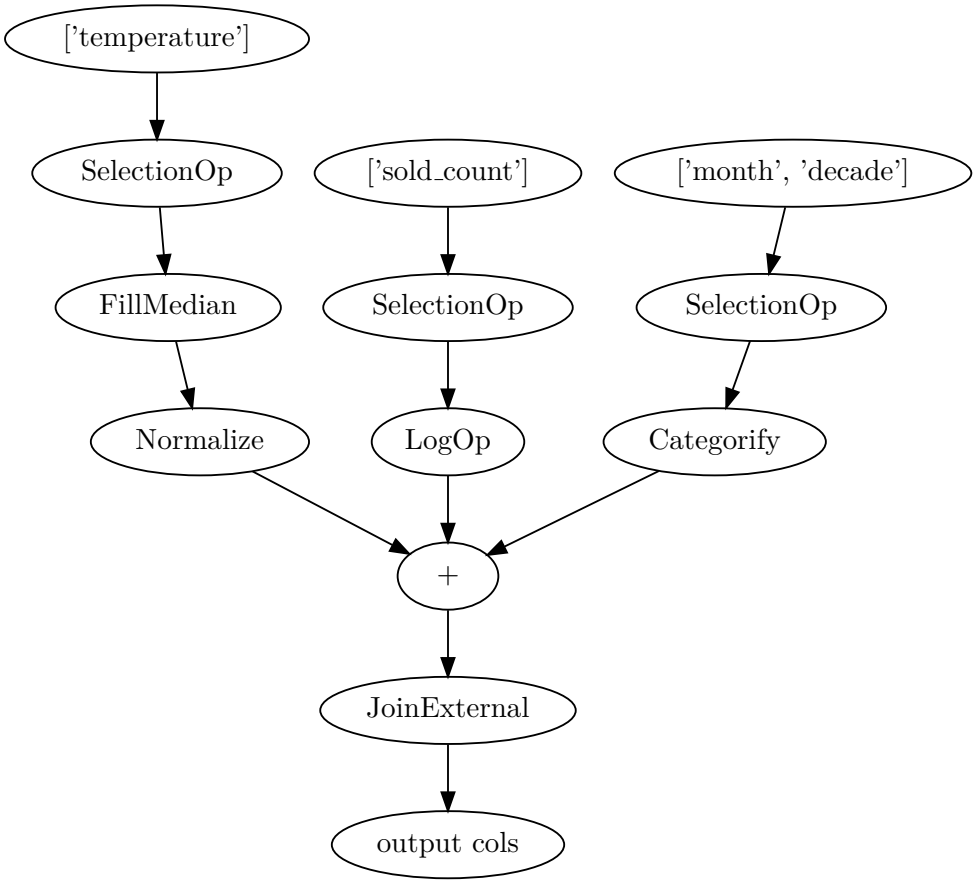

Figure 2.2: Example Workflow

other words, Operator can perform it's function without metadata about the whole dataset. StatOperators use metadata that need to be calculated before the operator is applied. Let us say we would like to min-max normalize a column. Min-Max normalization is defined as follows.

$$y_i = \frac{x_i - \min x}{\max x - \min x} \tag{2.1}$$

Where x is a column, x_i is the value of the row i in column x. To calculate this value, we need to know the minimal and maximal value in the column, in other words, a statistic. This is achieved by calling the function *fit()* on our Workflow and has to be done before actually applying it to the data.

Both kinds of operators offer a base class that can be inherited when we want to implement a custom dataset operation. The library also offers some of the commonly used operators already implemented, like Bucketize, Categorify, and Normalize.

2.3.1 Workflow

In this section we discuss the Workflow object interface and how to save and load a Workflow from a file.

Workflow can be instantiated by passing any Workflow Node to the constructor. Usually, we will want to pass the last Workflow Node of our Workflow graph, because we want to include all operations defined in the Workflow. Two main functions provided in the workflow object are *fit()* and *transform()*. This is similar to scikit-learn, a commonly used machine learning Python library. The functions *fit()* needs to be called before *transform()* only when the Workflow contains a *StatOperator*. When we violate this rule, the *transform()* method throws an exception.

The transformed dataset can then be saved to a directory with files that describe the data, operations applied to it, file names of the files the transformed data were spread across, and the data files. The only currently supported data format is Apache Parquet, which will be discussed later in section 2.4.1. This can be achieved by calling the *to_parquet()* function.

Any metadata gathered while fitting the Workflow will be used later during inference. Example of this is categorization. When categorizing values, we certainly want to remember what numerical value did we assign to a specific string value of a column, so we can reproduce this assignment later. This is why we also want to save the workflow. After calling the *save()* function on a workflow, NVTabular saves the Workflow object as a pickle. Pickle is a library for Python object serialization, meaning that the object representation is saved to a file and can be later loaded back into memory. Metadata such as Python version, cuDF version, and NVTabular version are stored as well as previously mentioned *StatOperator* data.

2.3.2 Column Selector

This section describes Column Selectors and how they are relevant to the process of designing a NVTabular Workflow.

Understanding Column Selectors is critical to designing our NVTabular workflow. The sequence of operations applied to data can be shared by multiple columns. Let us consider a dataset, where two columns have string values saved in them. One corresponds to a user's highest degree of education, like elementary school, high school, and college. The second column stores data about the users age group, like child, teenager, adult, and senior. Both of these columns data incorporate a sort of order, and the cardinality of values is low. This makes for a perfect example to use a Categorify Operator, see 2.3.3, on both columns at once.

These column selectors are defined independently of loading the dataset into a memory. Column Selector can be instantiated explicitly, by using the class constructor. This constructor takes in a list of strings, the names of

Figure 2.3: Example operator use

```
import nvtabular as nvt
categorical_cols = ["age_group", "education"]
categorized = categorical_cols >> nvt.ops.Categorify()
```

the columns we wish to select. Column selector constructor is rarely used in practice, as there are overloads implemented, so we can work with a list of strings the same way we would with a column selector.

2.3.3 Operators

This section is dedicated to Operators and their base classes offered by NVTabular. We will describe some of the available, already implemented operators and their parameterization. We will refer to NVTabular Operators with uppercase O or by name. Native Python operators are referred to as operators with lower case o.

Operators are applied to either a Column Selector or Workflow Node. Adding an Operator to a workflow is achieved by using the overloaded bit-shift operator », as shown in Figure 2.3. This bit-shift operator can be applied either to a Column Selector or a Workflow Node and returns a Workflow Node. Multiple operators can be chained together to reach the desired result.

2.3.3.1 Operator Base Classes

Base classes for operators, namely Operator and StatOperator aim to make the implementation of custom Operators easier, and the code more readable. At the time of writing this thesis, these implementations and how to derive from them are unfortunately not documented, and experimentation has not led us to any results worthy of attention.

2.3.3.2 Categorify

Categorization in this context is the mapping of any values to a range of integers. Categorify is the name for the Operator that implements the commonly used but configurable encoding. Now we explain some of the configuration options available to us.

Frequency threshold is a threshold that dictates how many occurrences of a value need to be present in the dataset within a specific category, for the category value to be included in the categorization process. This is useful when we spot some outliers in the dataset, meaning we can exclude rare category occurrences from the process.

Categorify Operator can be configured to use hash encoding. This means that there is a hashing function included in the transformation process. Hashing function can be modified with modulo, to define the maximum feature cardinality of the result. When we do not want to use hash encoding, Categorify uses label encoding.

"The main advantage of using Hash Encoding is that you can control the number of numerical columns produced by the process. You could represent categorical data with 25 or 50 values with five columns (or any number you want)."[11, section 3]

2.3.3.3 Normalize

Normalize Operator performs transformation so that the resulting data will have a mean of 0 and standard deviation of 1. This process is also known as standardization. For normalization to the range of 0 to 1, we can use the Min-Max normalization, described in Figure 2.1. This is achieved by using NormalizeMinMax Operator.

2.3.3.4 JoinExternal

Joining is a typical operation performed when our data is split into multiple files. We then use a unique identifier to join the data, so we get a data set containing all the required information we use. JoinExternal allows us to join external data sets, allowing us to choose between left and inner join. We can also specify the names of the columns by which we want to join.

2.3.3.5 LambdaOp

LambdaOp is an Operator that takes a lambda function as a parameter. This lambda function can apply any operation row-wise, meaning that at the time of execution, the lambda only works with information contained in a single row. We can also specify a list of dependency columns. This is useful when we refer to a different column value in the data set from within the lambda.

2.3.4 Data Loader

Data loading and data access are the main parts that Merlin focuses on in its optimizations. Not having a steady stream of data from disk into GPU memory is one of the bottlenecks when processing the data or training a neural network. Compared to TensorFlow, NVTabular Data Loaders offer the following features and optimalizations:

- Loading the data from disk batch by batch instead of item by item
- Processing data set larger than GPU memory

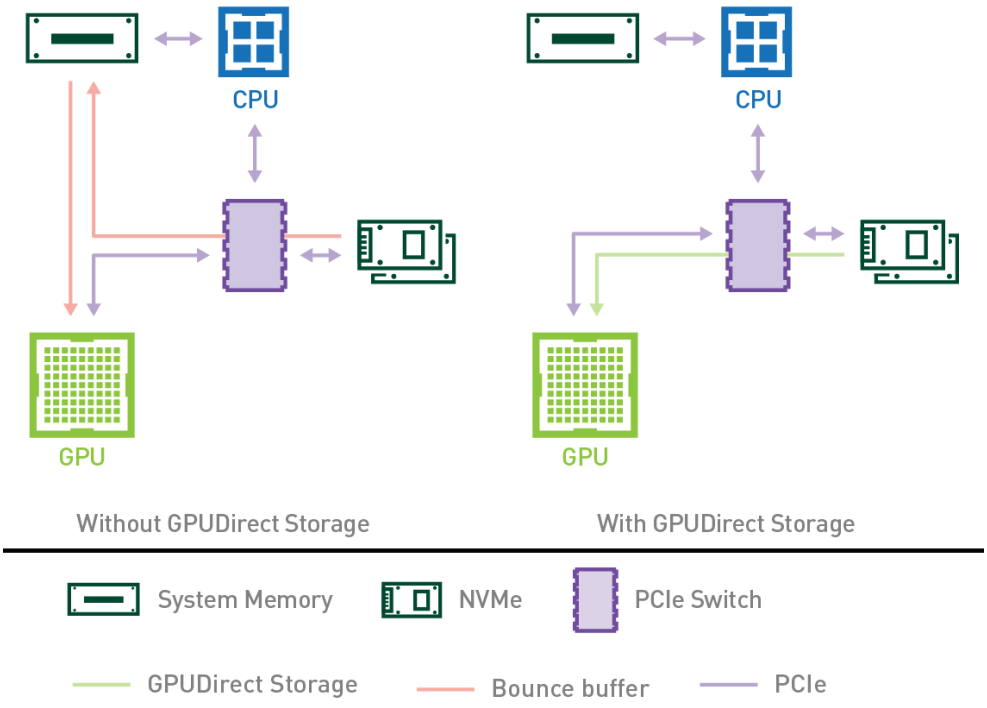

Figure 2.4: GPU Direct Memory Principle[2]

- Eliminating CPU-GPU communication, loading the dataset directly into GPU
- Supporting binary file formats, such as Parquet
- Tensorflow Data Loader interchangeability

The most important part here is *loading the data from disk directly to GPU memory*. Usually, the CPU dictates what data need to be loaded. The data is then transfered to system memory (RAM) and only then, passed from system memory to GPU memory. With the development of the GPU industry, the GPUs became so fast that supplying the GPUs with enough data to process became the bottleneck, rather than GPU computing time. This process has been redesigned and with the help of *GPUDirect Storage* (GDS) the data travel from the storage directly to the GPU memory, as illustrated in Figure 2.4. This helped both reduce I/O latency and increase memory bandwidth. I/O latency matters the most when a program requests new data. When new data for a GPU is requested, neither CPU nor RAM is bothered with communication, which leads to improved latency. The latency reduction is visualized in Figure 2.5. Finally, the overall memory bandwidth, which corresponds to

the amount of data that can be loaded into the GPU memory per second, increased up to two times, as shown in Figure 2.6.

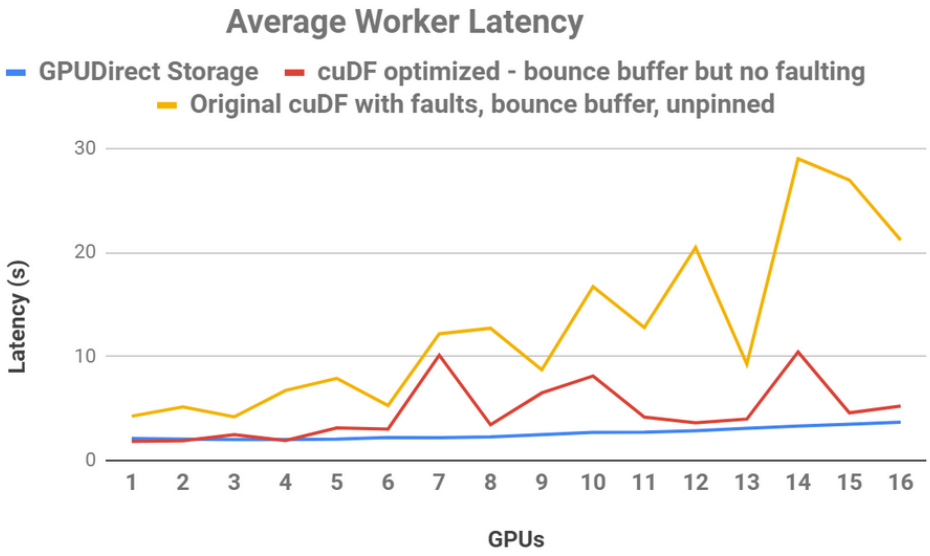

Figure 2.5: GPU Direct Memory Latency[2]

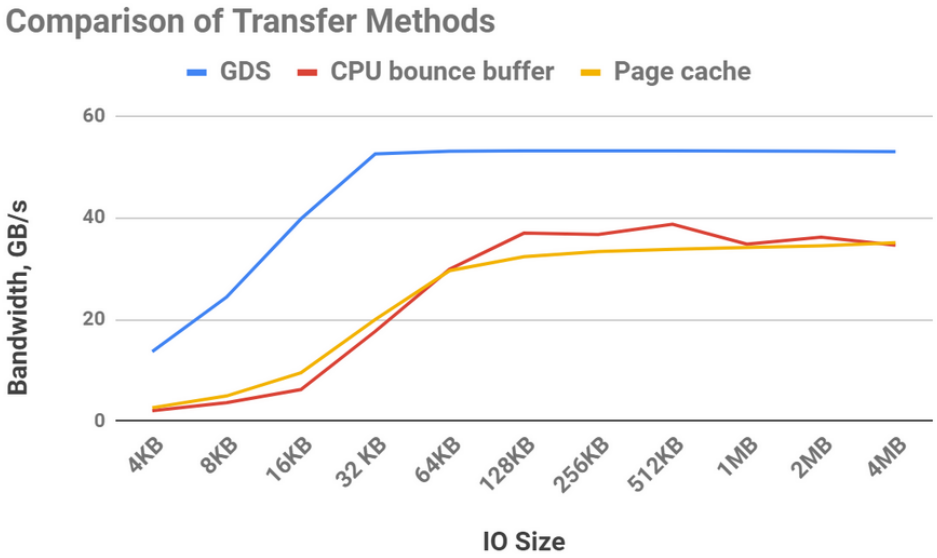

Figure 2.6: GPU Direct Memory Bandwidth[2]

NVTabular data loaders are parametrizable. Multiple implementations have been created to support Tensorflow and PyTorch interoperability. We

will now address *KerasSequenceLoader* and *KerasSequenceValidater* implemented in NVTabular to see how they can substitute their counterparts provided by Tensorflow.

Both *KerasSequenceLoader* and *KerasSequenceValidater* are fairly simple to use. *KerasSequenceLoader* takes in as parameters the data files or the data set in memory, column description, batch size, buffer size, and shuffle options. Column descriptions consist of separating the columns into categorical, numerical, and label based on column name. Batch size is the size of the data prepared for training at a time. The buffer size is the percentage, or exact number of GPU memory maximal consumption. The size of the buffer serves the purpose of loading multiple batches ahead of time. Shuffle parameter is a boolean and it states whether to randomize the data before sending it to training or not.

2.4 Data File Formats

In this section, we look at how different file formats can affect the performance of I/O operations. We also describe the Apache Parquet format in detail.

The available file formats can differ in readability, compression, nesting, and tool compatibility. The most common and widely used formats, such as csv or json, are relatively simple, readable by humans, and widely supported in the IT industry. When we take a look into the cuDF documentation, we discover multiple supported formats to load and save a DataFrame. These include the mentioned csv and json, but also parquet, orc, hdf, and feather. We will discuss the parquet format in more detail in the following sections, as it is supported by Merlin.

One big difference that sets file formats apart is whether they are *row* or *column* oriented. Row formats store data row by row. Columnar data formats store first all (or a partition) rows' first column data, then the data for the second column. This distinction comes from the fact that different workloads access data differently. This is best illustrated with the following scenario. Let us say, we want to print a movie with all its features and properties to the screen. We have to look up where the movie is located in the file and then load all the movie information into memory. This is a row-oriented workload. Now let us consider a different, more data-science oriented yet simple scenario. We want to calculate the average movie length for all movies. For this calculation, all we need to know is the length of the movie, other movie metadata do not interest us. This makes a columnar data format very well suited for us, as the storage can read data continuously and does not need to perform a lookup as often, nor does it need to read data that is not relevant for the calculation.

Another difference is the format of the saved data itself. There are plenty of formats available, but for our use case the main difference will be in size.

During data pre-processing and model training (discussed in Section 2.5.1) the data need to be read from the storage and then manipulated. The physical size of the data on a storage clearly plays a big role in how fast we can go through the data. Two files containing the same amount of information can differ in size. If we want our data to be readable by humans who only use a basic text editor, we save the data to a text file. This file can contain strings and numbers, but all these numbers are saved as text as well. This is because computers nowadays support saving and reading a plethora of characters, and so our encodings have to support thousands and thousands of different characters as well. If we agreed to use a special tool to read and manipulate the data, we could easily save space. Formats that require special tools, software or hardware to read and manipulate data are called *binary* file formats. The binary formats being non-readable allow for space optimizations, and this is useful, especially when handling large amounts of data. This is one of the optimizations that the Merlin framework tries to utilize.

2.4.1 Apache Parquet

In this section, we present Apache Parquet, an open-source binary data format developed for use in Hadoop big data ecosystems. Support for parquet is built into the Merlin framework and so we will take a look at how does the format work and what it supports.

Apache Parquet file format is visualized in Figure 2.7. The Parquet file format supports both simple and complex nested data types. It is designed to store booleans, integers, maps, lists, and the like. The file is split into *Row Groups*, which are essential file chunks. We can imagine a Row Group as a table, where each row contains single column data for a number of rows in the original data set. After the column data, the metadata segment follows. The second row contains data from the second column and the corresponding metadata. Single Row Group contains data from all columns, but each column is stored contiguously. Along with multiple Row Groups, each file also contains metadata for the file as a whole.

PyArrow is the Python library for manipulating parquet files that Merlin leverages. In PyArrow, we can specify the Row Group size for the engine and customize the file ourselves. NVTabular even specifies that Row Group size of 128MB is optimal for its performance. According to [3, page 3] Row Group size when handling big data is typically larger, around 1GB.

The Row Group metadata even contain statistics, like the maximum and minimum value for each column, which sometimes allow the engine to skip a column chunk read. This slightly complicates the logic of the reading process, adding the need to check if it is necessary to read a column chunk in order to receive the required information. Such optimization, in turn, leads to performance boost.

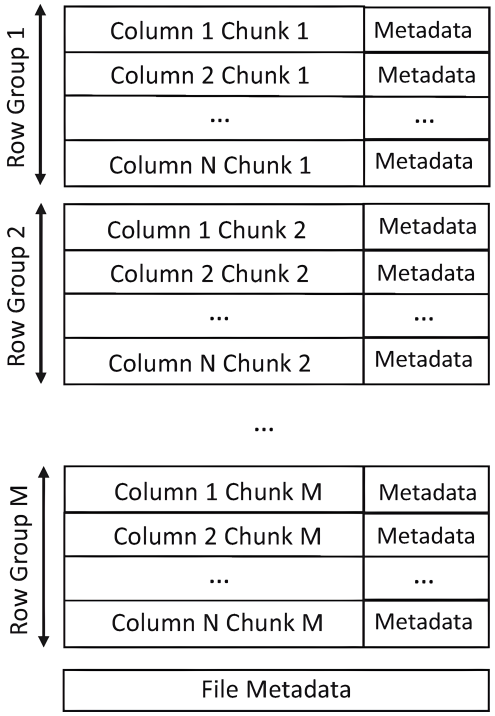

Figure 2.7: Parquet File Format[3]

Compression algorithms can also be used when saving the data in Parquet format. Compression takes place per chunk, so whole chunks can be skipped without decompressing the data. Some of the supported codecs are GZIP and Snappy.

2.5 Machine Learning Model

In this section we discuss what machine learning and especially deep learning is. Then we describe one of the modern approaches used in recommendation. Then we discuss how a model is implemented in Tensorflow, one of the most commonly used tools for deep learning. Merlin also provides a framework for building and training deep learning models called HugeCTR, which will be discussed as well.

Machine learning is a term that describes any sort of algorithm that can learn based on provided data. This learning process usually aims to create or adjust a model. This technique proved to be useful in many different fields and industries. Machine learning models are created for a wide variety of purposes, such as classification, regression, clustering, etc. Examples of those in practice would be a recommendation system, facial recognition, image classification, and natural language processing. The most promising subset of

machine learning is deep learning, so deep learning experienced the greatest boom, influx of people, and changes in the past few years. Deep learning refers to training an artificial neural network with multiple hidden layers, usually three or more.

2.5.1 Artificial Neural Network

Artificial neural networks are oversimplified models of a human brain. However, they are still far too complex for us to interpret. Artificial neural networks (shown in Figure 2.8) are sometimes referred to simply as neural networks (NN). NNs are a sort of complex mathematical functions that try to approximate a function that best describes the problem presented. They are a composition of basic units called neurons. One or more neurons form a layer. A neuron is a function that receives inputs, weighs them with a weight corresponding to each of the inputs, and then calculates the sum of the weighted input. The result of a neuron is called activation (potential). Weights are numbers, and their value describes the impact that a previous neuron (or neurons) has on the activation of the following neuron. The activation of a neuron can only be calculated based on a specific input. In contrast, the weights are stable (unless we are training the model), and the values of the weights are what define how a model performs.

The input to a neural network is usually multiple numbers accepted by an input layer. In a recommendation system context, the input numbers can represent item identification, item features, user identification, and user features, but also interaction features (e.g. time of the interaction). This data is then passed to the next layer, activating some of the neurons and leaving others inactive. The data is passed deeper and deeper (forward) in the NN, until it reaches the output layer. The output layer also consists of neurons and their weights and produces a result. An example of an output layer in a NN could be a single output neuron that predicts if a user would like to interact with an item. The value of this neuron is then passed to an activation function, which gives us the result, a single number, representing a positive, negative, or somewhat neutral interaction.

The first step in creating a NN model is to define a layer architecture. Different architectures are more suited to different tasks. After defining the architecture, the model is initialized. Initializing the model means that we have to define the weights of the model. The initialization of the weights can be random or use some of the more advanced techniques, such as the Xavier Initialization [12]. After initialization, the model is trained on a data set. Training a model is the process of adjusting weights, and thus adjusting the approximation function to better fit results observed in previously gathered data. This is why data collection is more important than ever, as deep learning turned out to be more proficient at solving problems that are too complex (time-wise or design-wise) for human-defined algorithms. Finding weights for

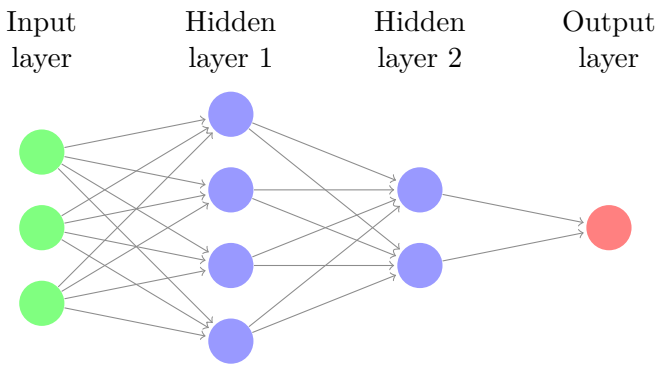

Figure 2.8: Example Neural Network

a model to produce good results is a lengthy process, as the input to the model is multidimensional data, and each of the weights contributes to the result. A model can easily consist of billions of weights, which are essentially parameters of the approximation function as well. When our model is trained, we can start using it to calculate the results. The process of receiving results from a model is called *inference*.

2.5.1.1 Embedding

When we lookup the definition of the word "embedding" in non-scientifical dictionaries, we find such phrases as *"to make something an integral part of"* [13]. In deep learning, embedding refers to creating an n-dimensional representation (a vector) of some values, such as texts, images, or even music. Representations of item features or words are assigned to a value in an n-dimensional space. This provides us with number representation of an object regardless of its type. Deep learning models only work with numbers, and therefore embedding can be critical to the process of feeding data into a NN. Another feature of embedding is that the resulting vectors exist in a space where values such as difference (or distance) can be calculated. How do we find the formula for mapping an arbitrarily complex item to a vector? What should be the distance between the red and blue colors? The answer is machine learning. Embeddings are also very important in Natural Language Processing, where it plays a big role in deducing semantics from words, as words can differ slightly in form but represent the same thing. One of the widely known and used algorithms that transforms word into embeddings is *Word2vec* [14]. As embedding can be interpreted as a machine learning task, the proper embedding calculation for every possible value is sometimes calculated beforehand, and the calculated embeddings are saved to an embedding table. Embedding tables, which are essentially lookup tables, can be very large, and thus, if we want to load them into a GPU memory and use them,

we need large amounts of memory available to us.

In Natural Language Processing, precalculated word embedding can usually be reused and shared among multiple NNs because the semantic meaning of the words does not change. In recommendation, however, we are often tasked with calculating our own embeddings. Some services, offering a large number of items to a user, will require a large embedding table, unless we want to always calculate an embedding during inference. Embedding NN takes a long time to train as any NN, and therefore using precalculated embeddings also saves us time. NVIDIA's deep learning framework HugeCTR addresses this problem with Embedding Training Cache, and we will discuss it in the matching Section 2.5.3.

2.5.1.2 Wide & Deep

Wide & Deep is a NN model architecture, presented by Google Inc. in a paper[15] published in 2016. Wide & Deep, as the name indicates, is a NN with two parts: wide and deep. Before this article, wide models and deep models were usually used independently, and this architecture tries to utilize each model's strengths. The strength of deep models is to find deeper relations between the features of the items. The strength of wide models in Wide&Deep architecture lies in memorizing what items go well together. The wide part of the model is what is considered as an alternative to logistic regression, which is another machine learning algorithm that is sometimes used when recommending. Google has been able to use this relatively new Wide&Deep architecture to increase application acquisitions on their service Google Play. Google Play relies on the recommendation system when presenting the user with applications selected from thousands of possible candidates.

The wide part in this model plays the role of identifying what items go well together. On the input, the wide part receives a history of items liked by the user (their identifiers) along with an item identifier for the predicted identifier. The deep part of the model receives all the items' features along with its identifier. Multiple hidden layers in the deep model are usually interpreted in such a way that the deeper the layer, the more complex the feature of the item each neuron represents. An example of Wide & Deep model architecture is shown in Figure 2.9.

2.5.2 Tensorflow

Tensorflow is a widely used deep learning framework. It is essentially a collection of modules that provide the tools required to build and train deep learning models, manipulate datasets, and more. Merlin has built-in interoperability with Tensorflow, so we will take a look at how TensorFlow fits into the Merlin pipeline. Tensorflow provides Feature Columns, a module to help with conversion of raw data into data digestible by a NN. Keras is a library built

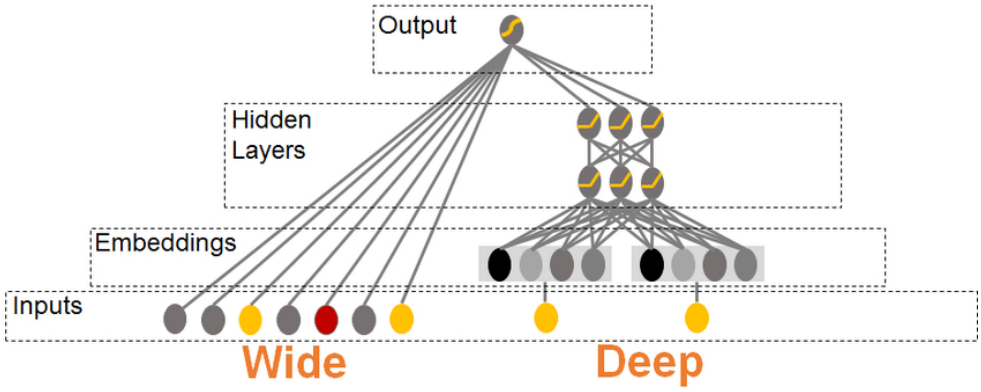

Figure 2.9: Wide & Deep Model Architecture[4]

on TensorFlow. Keras used to be a separate framework, supporting multiple underlying machine learning libraries among which was TensorFlow. Since TensorFlow version 2.0 Keras is fully integrated in TensorFlow and does not support other machine learning libraries. Keras provides an API, for creating machine learning models by layer composition, which is a more high-level view than what Tensorflow allows. Keras also contains metrics and optimizations that have already been implemented for model training and evaluation.

2.5.2.1 Feature Column

Feature Column is a class in Tensorflow, meant to be used on raw data. Feature Columns are mainly used to specify how an input to a NN is going to look. There are multiple transformations that are applicable on the raw data and we take a look on a few along with examples in the following section. The Feature Columns are then passed into Keras Input layers.

The features can be split into two types: categorical and numerical. Categorical features usually have low cardinality in contrast to numerical features, where a value for each row in the data set can be unique. In the Tensorflow feature column namespace we find such a distinction, namely the Categorical column and the Numeric column.

Numeric columns are the simplest columns as they do not change the data at all. They simply pass the column values unchanged.

Categorical columns provide multiple ways of mapping raw input data into vectors of numbers that are digestible by a NN. Categorical columns are the equivalent way of treating unique data as One-Hot Encoding. One-Hot Encoding is visualized in Figure 2.10. One-Hot Encoding is a technique where we introduce a feature to the whole data set for every individual value of a categorical feature. This newly created columns value is equal to 1 when the original value corresponds to the newly created column and zero for all

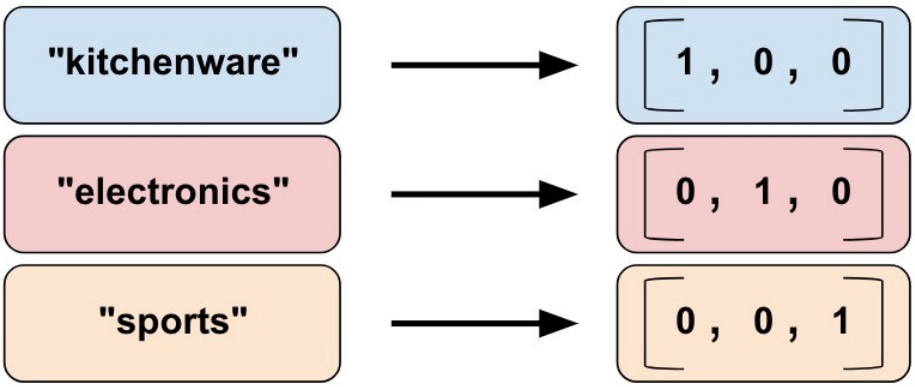

Figure 2.10: One-Hot Encoding[5]

other newly created columns. The process of One-Hot Encoding can increase the number of dimensions for the data, which results in more weights in the resulting model. More weights in the model result in longer training times, but also in capturing a more complex relationship between input data and result.

Bucketized columns can be applied to a categorical feature or a numerical feature. As the name suggests, the Bucketized column splits the data into buckets, based on criteria specified by us. An example of this could be splitting a year feature into four buckets, depending on the value.

Embedding columns are used when input categorical feature has high cardinality but is still a categorical feature. It would become unfeasible to apply One-Hot Encoding in such a case because the dimension of the data would grow drastically. This problem is known as the curse of dimensionality [16]. The embedding can map such values into a lower-dimensionality space. This comes with trade-offs, bringing into the model the concept of distance between two values, which may not be desirable.

2.5.2.2 Model

Model is a Keras class that encapsulates a model layer architecture. It is intended to group multiple NN layers that can be chained either sequentially or by using the newer Keras Functional API. As Keras' sequential models have multiple limitations, in this Section and throughout this thesis, only the Functional API will be discussed.

The Model class supports saving a model to storage and loading a model from storage. To save a model, Keras stores the model architecture definition, model weights (both of which are explained in Section 2.5.1), an optimizer, and a set of loss and metric functions. Loss is a function that measures the

difference (or distance) between the desirable result and the current result infered from a model. The optimizer's role in model training is to decide how to adjust the model weights so that the loss function is minimized.

When defining a NN architecture, we usually start with the input layers of the model. Input layers are created using the Input class in Keras and they serve the purpose of defining what attributes the input to our NN has. The input layer constructor takes the following parameters: column name, data type, shape, sparsity, etc. Data type specifies the data type of the input, typically integers or floating-point numbers. The shape dictates the input shape of the data. The input shape can be any tensor shape, where a tensor is basically a vector or a matrix of a specified shape initialized in the GPU memory. After defining the input layers, we usually create embedding layers that convert preprocessed input data to embeddings. The most important parameters of an embedding layer are the input dimension and the output dimension. When embedding layers are defined, we can attach fully connected hidden layers (dense layers) to create a deep model. The Keras Functional API, as the name suggests, is designed to work with layers as you would with functions. A layer in NN model takes as input the preceding layer, and this is no different in Keras, we simply pass the preceding layer to the following, and Keras handles connecting the layers. When we want our model to return a single number, we can then append a single last dense layer with the size of one, which will result in a single neuron layer.

After laying out the layer architecture, we simply pass all the input layers and the last output layer to a Model class constructor, and Keras handles the rest. When the model is initialized, it still needs to be compiled. Compiling a model requires specifying what optimizer, loss function, and metrics do we want to use and track. For information on how to choose an optimizer, what is a gradient descent, and how to choose an optimizer, please refer to this article [17] which serves as a basic orientation guide.

2.5.3 HugeCTR

HugeCTR is the second part part of Merlin after NVTabular. HugeCTR is an abbreviation for Huge Click Through Rate, implying its use in the area of building recommendation systems. It is an optimized machine learning library written in C++. HugeCTR is designed to support model training across multiple GPUs. HugeCTR's interface for defining a model's architecture is very similar to Keras. HugeCTR also provides hardware abstraction, allowing for scaling up from a single GPU to multiple machines or even clusters of GPUs. One of the optimizations required for GPU scaling is embedding table distribution. The embedding table distribution mechanism is called Embedding Training Cache and it comes with an in-built smart memory caching mechanism. Another optimization is Mixed Precision Training, which is utilizing NVIDIAs GPUs Tensorcores to boost matrix multiplication and also changing

some layers data type to a floating point number with only 16 bits representation. Many parts of HugeCTR can be configured, so we will stick to the high-level view, which will give us a rough idea of how the framework works.

A model can be defined in HugeCTR in a way very similar to Keras. The process of defining the model in Tensorflow Keras is described in Section 2.5.2.2. When training a model in HugeCTR, the framework also allows us to specify learning rate scheduling. The learning rate is an optimizer parameter that specifies a step size when adjusting the weights of the model to achieve the minimization of the loss function. Learning rate scheduling allows us to use a higher learning rate in the beginning of the training, and after specified number of epochs, the learning rate starts to decay.

2.6 Triton Inference Server

The Triton Inference Server is the last part in the pipeline and is currently recommended to use in Merlin. Triton Inference Server is a server for model inference, request batching, and model instance scaling. The server, much like all other parts of Merlin comes in a Docker container, along with all dependencies required for inference. We will now discuss what is required to run the server, supply a model, and succesfully infer results from a model. Triton Inference Server architecture is illustrated in Figure 2.11.

2.6.1 Model Repository

Model repository can be located on the local file system or cloud storage services, such as Google Cloud, Amazon S3, and Azure Storage. The model repository is made up of stored models. Triton supports loading models from multiple repositories at the same time. A saved model at this point is a directory with the name of the model. A model directory contains a model configuration file that contains information about the inputs, outputs, and model composition if the model is an ensemble. Ensemble is a term used for models that are made up of more models and then connected. When using NVTabular for preprocessing along with a machine learning model, the resulting model will always be an ensemble model. The model directory further contains versions of the model itself. When loading a model into Triton memory, we can specify which version of a model we want to use. Triton supports plenty of modern models created in frameworks such as Tensorflow and PyTorch, but also models in openly defined formats, such as ONNX or OpenVINO. For a more specific definition of the model format, please refer to the documentation [18].

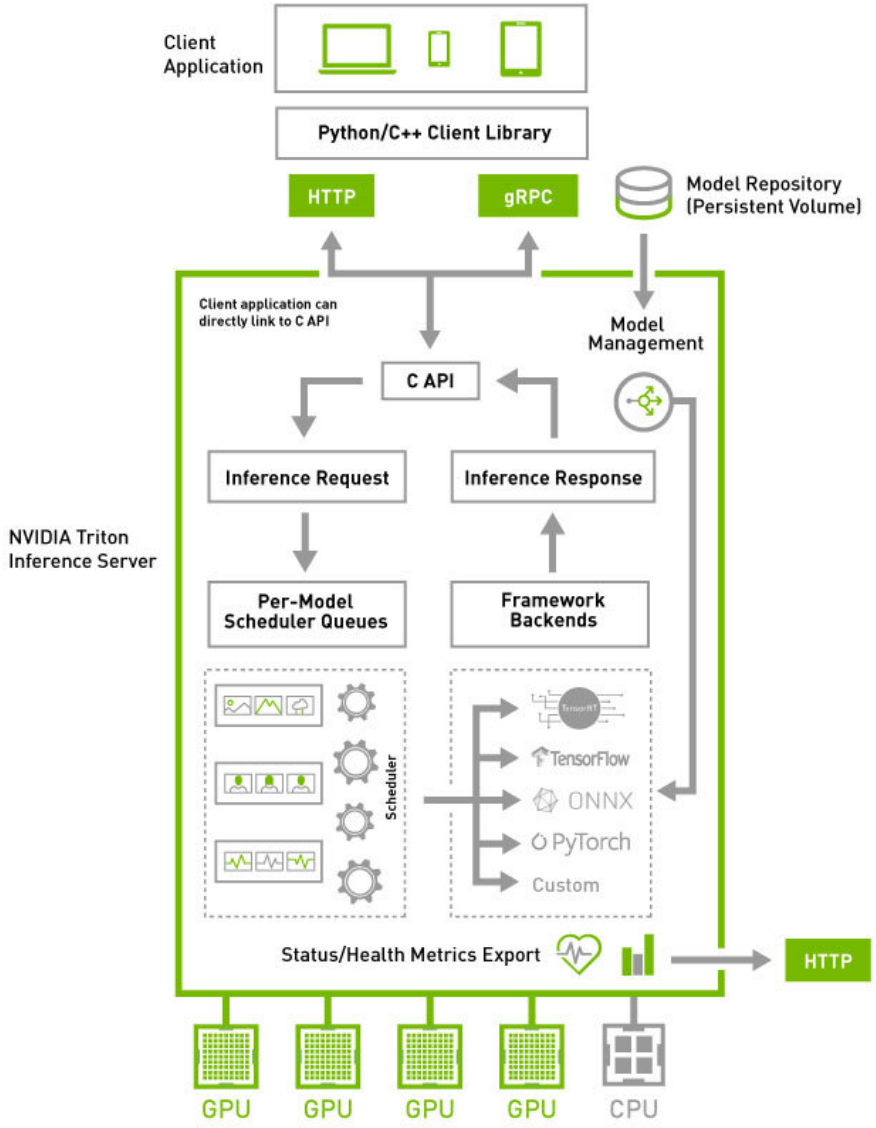

Figure 2.11: Triton Inference Server Architecture[6]

2.6.2 Requests

When Triton is started up, it automatically hosts two services to receive requests. First is a HTTP Inference request service which listens for HTTP requests and then queues them for Inference. The second is a Google Routing Protocol service that listens for GRPC requests. GRPC is similar to HTTP but introduces some optimizations to the communication process. Triton provides an API for making requests to both of these services. Establishing a connection to a Triton server means simply to instantiate a class with an IP address as a parameter. After creating an inference client, we can use it to probe the server to see if it is alive and if the connection is working. When our client is set up, we can make requests to the server, such as asking about available models, models configurations (required inputs and outputs), or providing inference statistics.

2.6.3 Model Ensemble

Triton server supports ensemble models. An ensemble model directory contains similar configuration files as described in Section 2.6.1. The configuration file of an ensemble model requires an additional section that defines *steps*. Let us consider an ensemble model composed of NVTabular Workflow and a NN created in Tensorflow. There would be two *steps* in such a case. The first step describes inputs and outputs of a Workflow. The second *step* describes the input and output of the NN model. Triton Inference Server uses this information for, among other things, inference from a single model of an ensemble.

Triton server contains a Python module that allows us to export an NVTabular Workflow and Tensorflow model into an ensemble. This is not limited to only TensorFlow but supports PyTorch models and HugeCTR models as well. The provided function takes in an NVTabular Worklow instance, Model instance (or path to model directory), output path, and label column name as required parameters. Other parameters we can specify are the model name, the names of categorical features, and continuous features.

2.6.4 Request Scheduling

Scheduling the requests that arrive at the server can help optimize inference and increase the throughput of the entire process. Triton server implements three different scheduling strategies. *Oldest* strategy is a strategy for grouping incoming inference requests based on a time window. When a certain time windows passes, all requests that arrived to the server during the time window are grouped into a batch and sent to the model for inference. *Direct* is oriented on models that keep a state associated to requests. This means that requests cannot be batched, as each client sends requests in a sequence. When using the Direct strategy, each request belongs to a single batch. The state is assigned

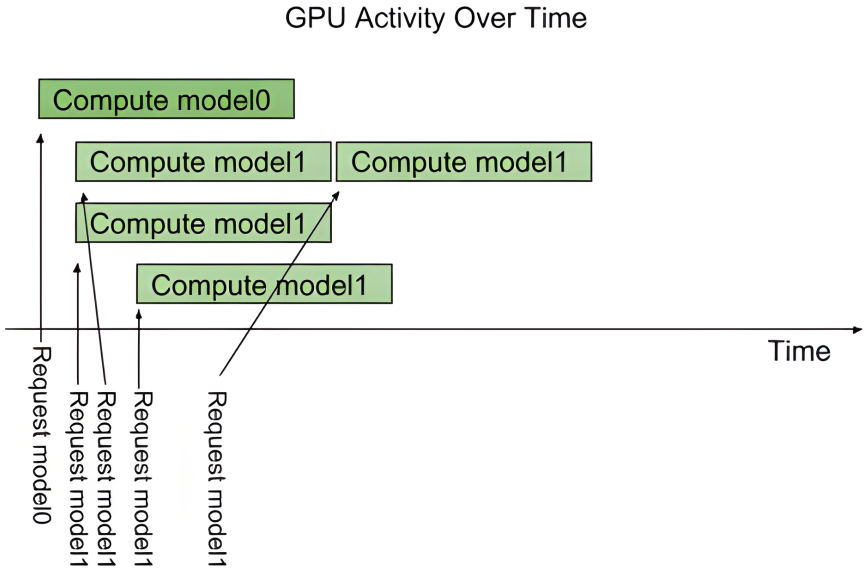

Figure 2.12: Parallel Model Execution[6]

to a batch slot. Finally, the *Ensemble* strategy is used when we try to infer from an ensemble of models.

2.6.5 Inference

When infering from a specific model loaded in Triton server, we need to prepare a few parameters. First a list of column names, the request will contain. Second a Pandas dataframe or cuDF dataframe, that is composed of columns listed in the first parameter. Third parameter when creating a Triton inference input, we specify a type (class) of the input. These parameteres are then encapsulated by an inference request object. This object is then ready to be sent to a specific model loaded in the memory of Triton server. Each inference request takes a model name as a parameter. Note that we do not specify any instances of the model, as this is handled internally by the Triton server, which has been designed with scalability in mind, like other components of the Merlin framework. Options for Parallel Model Executions are also available, if we would like to use multiple instances of the same model at the same time. Parallel Model Execution is illustrated in Figure 2.12. After the Triton server processes our requests, it returns an object that contains the result. We then simply extract the data as a tensor or numpy array and save it to storage.

Chapter 3

Realisation

Now that we are familiar with Merlin components, we will use this knowledge to build a recommendation system. First, we prepare our Docker containers for training and inference. We host a JupyterLab service for development and experimentation to take place in. Then, we choose and describe a MovieLens data set. After looking at the data, we will design a data pre-processing Workflow using NVTabular. For optimal performance, we use the Apache Parquet columnar data format, described in Section 2.4.1. Then, using TensorFlow we define a variation to the Wide & Deep neural network architecture and train the model. Training is monitored using Tensorboard. The trained model is then injected into our inference container and we measure throughput.

Throughout these steps, we take notes on what processes are demanding on GPU processing power and memory. We discuss the limitations and benefits of the framework as a whole, including documentation, stability, error reporting, and ease of use.

3.1 Environment

Taking the time to set up a proper development environment can be time consuming but usually pays off in the long run. NVIDIA manages their Docker containers for machine learning and other purposes on a site called NGC.

"*The NGC Catalog is a curated set of GPU-optimized software for AI, HPC and Visualization.*"[19]

3.1.1 Docker containers

First we download Docker containers Merlin-training and Merlin-inference. During the development process, we were forced to update our container images, as the newer versions contained bug fixes for encountered problems. All the final code discussed in this thesis is run on the 22.02 versions of the containers which were released in February 2022. For those trying to reproduce

the results, we recommend downloading images with the same version, as updates can sometimes break old functioning code, and the Merlin framework is undergoing structural and functional changes at the time of writing this thesis.

Merlin-training is a container that contains NVTabular, HugeCTR, TensorFlow, Pandas, Numpy, cuDF, and other essential Python packages we use during development. Before running the container we create a directory that will contain all notebooks, dataset, and models. After opening our working directory, the training container is run using the following command:

```
docker run
    --gpus all
    --rm
    -it
    -v $(pwd):/movielens
    -w /movielens
    -p 18888:8888
    nvcr.io/nvidia/merlin/merlin-training:22.02
    /bin/bash
```

We pass in all of the available GPUs to the container. After that, we specify we want to remove the container after execution is finished and run bash in interactive mode. We specify the image name along with a tag (version) and map the port 8888 in the container to 18888 on the host machine. In the container we start a JupyterLab service using the *jupyter-lab* command.

Merlin-inference is a container with Triton Inference Server and corresponding packages installed. The inference container is run using the following command:

```
docker run
    --gpus all
    --rm
    -it
    -p 18000:8000
    -p 18001:8001
    -p 18002:8002
    -v ${PWD}:/model/
    nvcr.io/nvidia/merlin/merlin-inference:22.02
```

This container has to be run from within the model repository, as it maps the currently opened directory to the *model* directory inside the Docker container. The *model* directory is the default directory for a local file system Model Repository (discussed in Section 2.6.1) used by Triton. In this case, we

map three ports that correspond to HTTP inference service, GRPC inference service, and health and metrics service respectively.

Later on, we will make inference request from the JupyterLab notebook running in the training container to the Triton Inference Server running in the inference container. Since we want the containers to communicate, we need to check what IP addresses are assigned to what containers. We can inspect the Docker bridge network. Bridge network is the default network Docker assigns the containers to as long as we do not specify otherwise. The command that shows us the containers IP addresses is:

```
docker network inspect bridge
```

After running this command we simply take a note on what IP address was assigned to the Merlin-inference container, as we will need to know it when creating an inference client.

3.2 Dataset

One of the datasets we have experimented with is the MovieLens 20M Dataset[20]. The data contains some basic movie metadata like genres and name. User interaction data store the value assigned by a user to a movie, as in rating. This dataset will be refferred to as MovieLens 20M.

The other datasets we have worked with, provided by the supervisor is private and the details will not be shared. This dataset will not be refferred to, but served us to build an initial pipeline.

3.2.1 MovieLens 20M

MovieLens 20M is commonly used in the recommendation community. It serves well when demonstrating a proof of concept or comparing model performance. It is one of few publicly available free large datasets.

The two files we will use are *movies.csv* and *ratings.csv*. The file *movies.csv* contains three columns: movieId, genres, and title. We have decided to remove movie titles from the dataset. Genres column contains lists of strings, the names of the genres. MovieId is a movie unique identifier shared across the two files. The file *ratings.csv* contain four columns: userId, movieId, rating, and timestamp. We have decided to remove the timestamp column, as we encountered problems trying to correctly process this continuous feature. Rating column contains a rating a user explicitly assigned to some subset of available movies. Rating values range from 0 to 5 with the granularity of 0.5 and they correspond to the typical movie star rating.

Dataset	Size on Amazon S3	Query Run time	Data Scanned
Data stored as CSV files	1 TB	236 seconds	1.15 TB
Data stored in Apache Parquet format*	**130 GB**	**6.78 seconds**	**2.51 GB**
Savings / Speedup	**87% less with Parquet**	**34x faster**	**99% less data scanned**

Figure 3.1: Parquet and csv formats comparison[7]

The dataset contains a total of 20 million ratings, assigned by 138 000 users to 27 000 movies. The original file sizes are 1.3MB for movie.csv and 650.7MB for ratings.csv.

3.3 Pandas

The columns timestamp and title were dropped before feeding the data to NVTabular, as it does not provide an Operator for dropping a column. After dropping the columns, we also sort the values in the genres column and then join the list values together using a comma. The reasoning will be explained in Section 3.4.

Movie data as well as ratings data are then saved to Apache Parquet format on the storage. The conversion from csv format to Apache Parquet reduced the movie file size from approximately 700KB to 200KB. This conversion alone saves us around 70% in storage space. These are the benefits of binary data formats discussed in Section 2.4, not to mention the data access times. An article[7] comparing the csv and parquet formats found similar results, along with other speedups and savings which are illustrated in Figure 3.1.

The data is split to train and validation subsets. Before splitting the data, we shuffle it to remove any ordering that might have been present in the dataset. The train data subset will make up 80% of the whole dataset, and the validation data subset will be the remaining 20%.

3.4 NVTabular

Now that our data is saved in Parquet we can move on to NVTabular preprocessing. The first utility provided by NVTabular is selection of a DataFrame library. DataFrame is the name both Pandas and cuDF use to denote an in-memory dataset. NVTabular provides a function that checks for available packages and if cuDF is present, it chooses cuDF, in other cases it falls back to Pandas. This behavior is what we observed when experimenting, unfortunately the documentation for the function is missing. The import of a DataFrame library then looks as shown in Figure 3.4.

Figure 3.2: Dataframe library import

```
# Get dataframe library - cudf or pandas
from nvtabular.dispatch import get_lib
df_lib = get_lib()
```

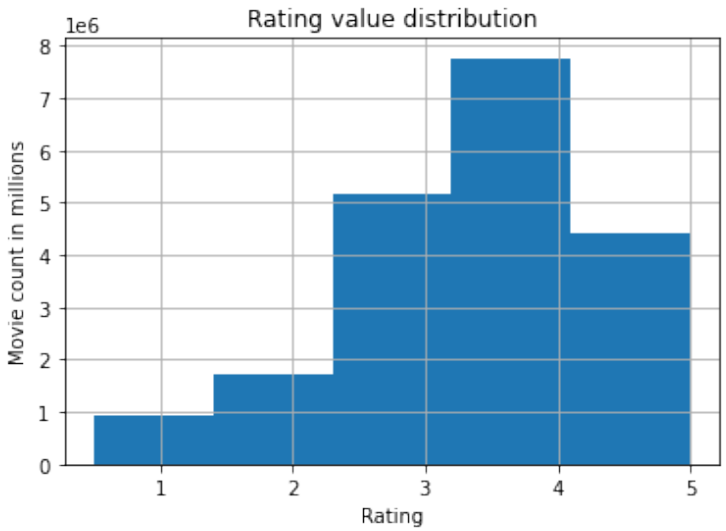

Figure 3.3: Rating Values Distribution

Now let us discuss the data preprocessing. First we will decide on which columns will have the same preprocessing pipeline. We will want to label encode the genres column. The genres column contains comma separated values of the movie genres. Label encoding the genres column will result in every unique genre combination being encoded as a unique number. UserId and movieId columns will also be label encoded, as this will map the values to number ranging from 0 to C-1 where C is the cardinality of the column. Label encoding in NVTabular can be achieved by applying the Categorify Operator. This means that the columns userId, movieId, and genres will have the same sequence of Operators applied to them.

Second let us take a look at the label column. Label is a denotion of the column we will be predicting, in our case, the column is rating. Rating column values range from 0 to 5. We will split the dataset based on movie rating, and consider a rating greater than 3 to be positive and the rest negative. Positive ratings will be assigned a value of 1 and negative a 0. This approach was proposed in a publication [21]. For this we will use the LambdaOp. The rating value distribution is shown in Figure 3.4.

Third we address the problem of the data being split into two separate files. We have our movies file and our ratings file. We need to perform

Figure 3.4: Workflow Definition

```
CATEGORICAL_COLUMNS = ["userId", "movieId"]
LABEL_COLUMNS = ["rating"]

joined = CATEGORICAL_COLUMNS
    >> nvt.ops.JoinExternal(movieDataFrame, on=["movieId"])
encoded_features = joined >> nvt.ops.Categorify()

ratings = nvt.ColumnGroup(LABEL_COLUMNS)
output = encoded_features + ratings
    >> nvt.ops.LambdaOp(lambda col: (col > 3).astype("int8"))
workflow = nvt.Workflow(output)
```

a join operation on these two datasets. NVTabular provides an Operator JoinExternal, which is used when we need to join two datasets on a column. Both files include the columns movieId. Let us join the data on the movieId column to receive a dataset composed of all of the following columns: userId, movieId, genres, and rating.

The finalized code is shown in Figure 3.4. The NVTabular Workflow resulting from such code is visualized in Figure 3.5. Next, the Worklow is fitted on the data. The fitted Workflow is then applied to our dataset and the transformed data saved for use in training. The Workflow itself is also saved to disk, for use when creating an ensemble model consisting of the Workflow and the model.

3.5 Model

We have decided to use TensorFlow for model definition because it is widely used by machine learning specialists. Experimenting with Merlin TensorFlow interoperability could potentionally be a rewarding process, as this could allow us to utilize Merlins strengths in already implemented recommendation systems. Before creating the architecture of our model, we load the Workflow from the disk and pass it as a parameter to the *get_embedding_sizes()* function that returns the suggested embedding dimensions for each feature. We will use these suggested embedding sizes later, when we create embedding layers in the wide part of our model.

We have designed a variation on the Wide & Deep Model. Our goal for the model is to predict the value of users interaction with an item. The model will return a number in the range from 0 to 1 expressing the positivity of a user item interaction. Just like when preprocessing, the number one will represent a positive interaction and the number zero will represent a negative interaction. The model can also return values close to 0.5 which will mean that the model does not have a decisive prediction on a user item interaction.

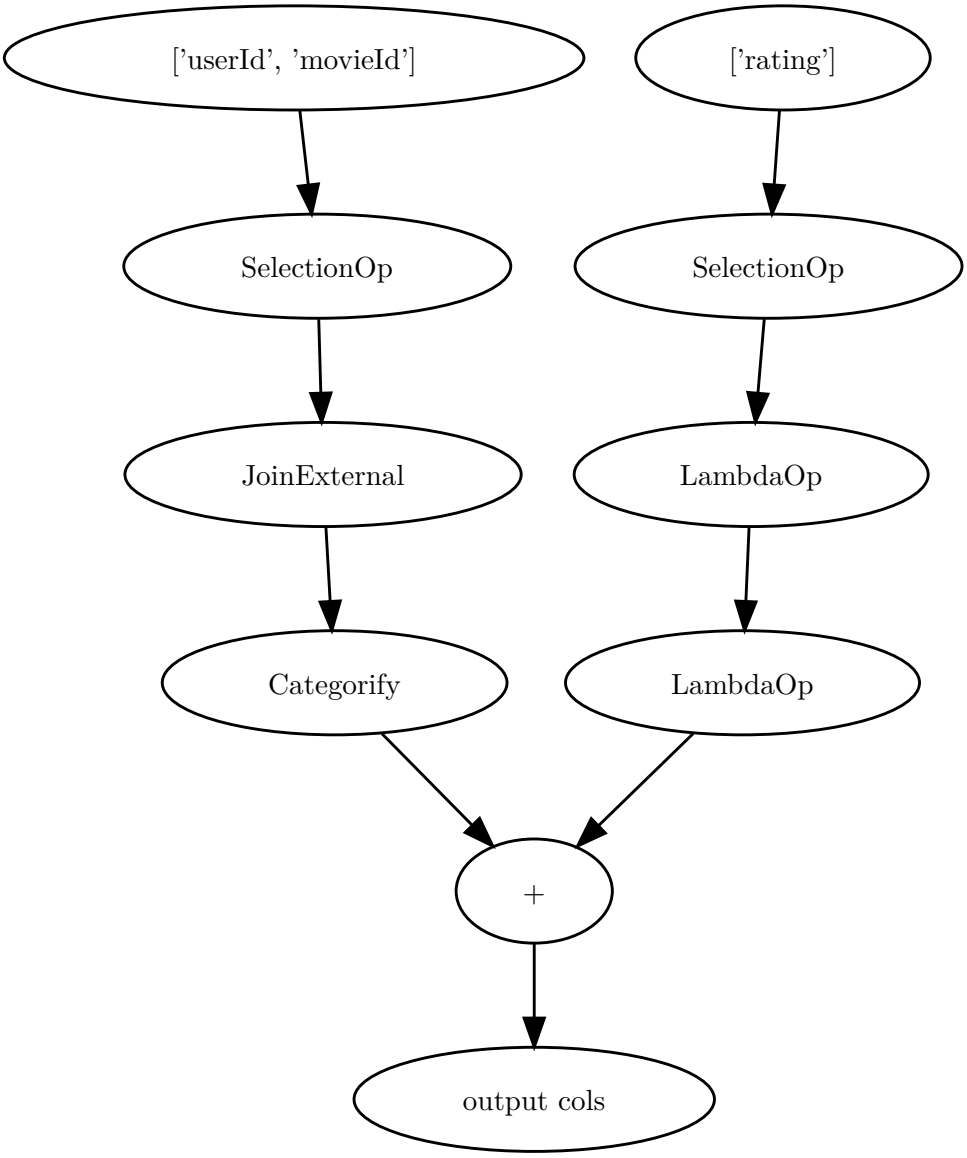

Figure 3.5: Example Workflow

The deep part (visualized in Figure 3.6) in our design will be accepting all of the available features, which are userId, movieId, and genres. In the deep part we try to utilize NVTabulars DenseFeatures layer, that automatically handles transformations for both continuous and categorical columns. After these DenseFeatures layers, we will attach the same sequence of layer three times. The layer sequence will consist of a Dense layer, Dropout layer, and BatchNormalization layer. The Dense layes will be composed of 32, 16, and 8 neurons respectively as we go deeper in the model. Finally, we append one last Deep layer with a single neuron, because we want the result to be a single number.

The wide part (visualized in Figure 3.7) of the model will utilize all of the available features as well. In the wide part, we start by creating TensorFlow Feature Columns, namely the IdentityCategoricalColumn for each of the input feature. These Feature Columns simply return an identity of the inputs, and are passed into Keras Input layers. The Input layers will be each followed by an embedding layer. After the embedding, we apply the Flatten layers, which changes the output shape of the embedding layers. A sum of the results returned by Flatten layers is a final result of the wide part of our model.

The final result from the model when the wide and deep parts are combined, is a weighted sum of the wide and the deep part.

3.6 Training

For model training, we need the to create a KerasSequenceLoader for loading and supplying the training data. The KerasSequenceValidater will be serving as a dataloader and a callback, and provide us with validation loss and other metrics during training. Another callback we will use during training is the TensorBoard callback, that also provides us useful information about how is the model training advancing.

The wide and deep parts of the model will be trained jointly. Joint training, in contrast to ensemble training, will optimize model parameters simultaneously. The joint training method allows for the parts of the model to complement each others' strengths. As wide models excel in memorization, deep models excel in generalization, using both combined makes for a promising model. The wide part of the model is evaluated using binary accuracy metric and our model reaches almost 80% on the validation dataset. The binary accuracy learning process is captured in Figure 3.8. The metric used upon the deep part of the model is the same as in the publication of Wide & Deep Model [15] and it is called Area Under Curve (AUC). In our model, it reaches around 0.86 as shown in Figure 3.9.

The loss function used during training was binary cross-entropy. This function is typically used when we are facing a classification problem, where

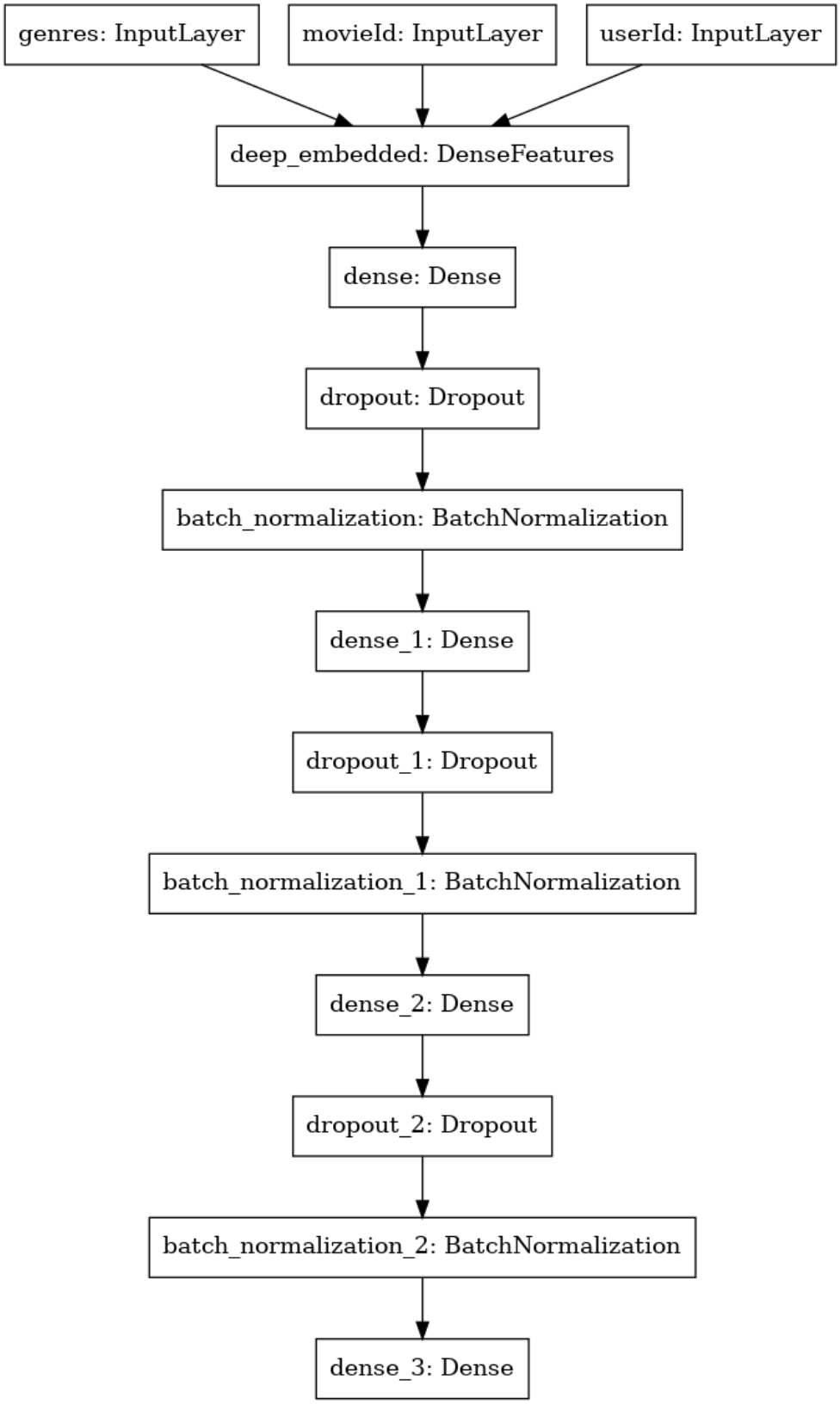

Figure 3.6: Our Models Deep Part

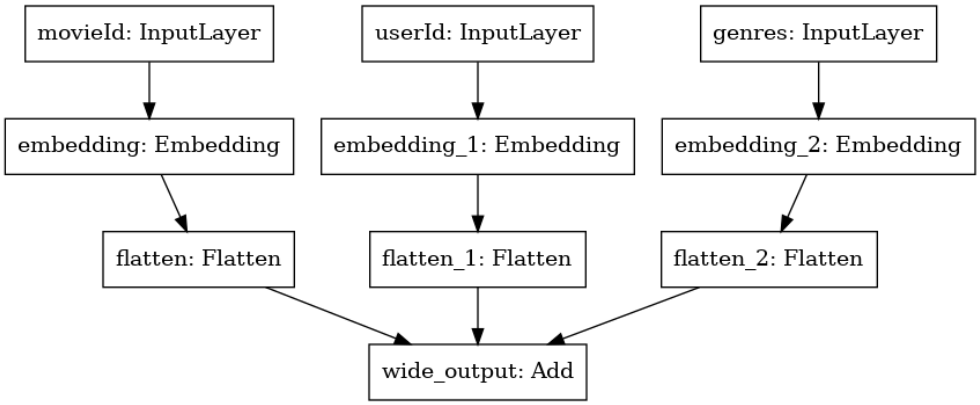

Figure 3.7: Our Models Wide Part

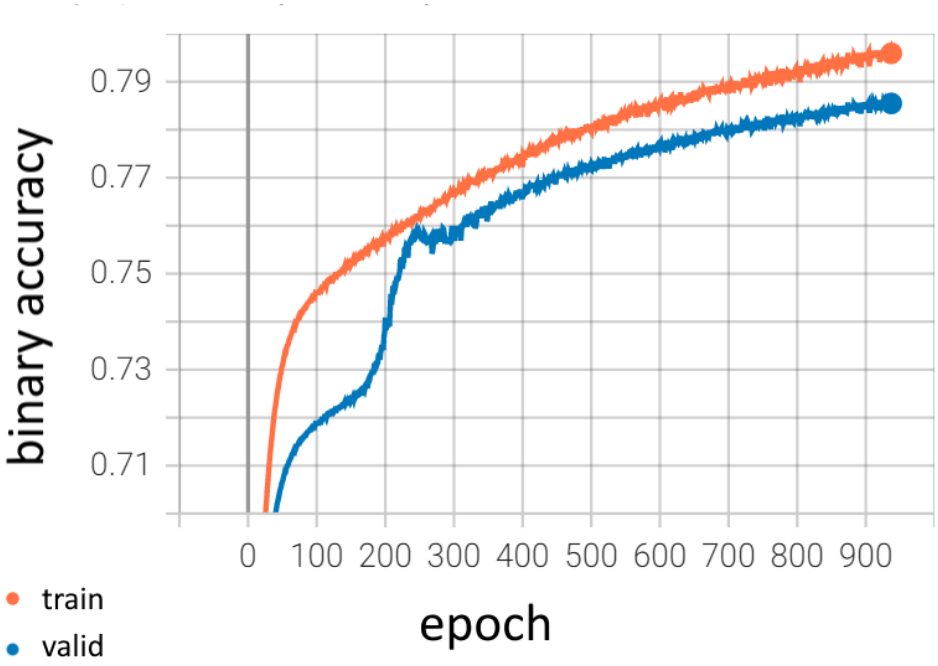

Figure 3.8: Binary Accuracy

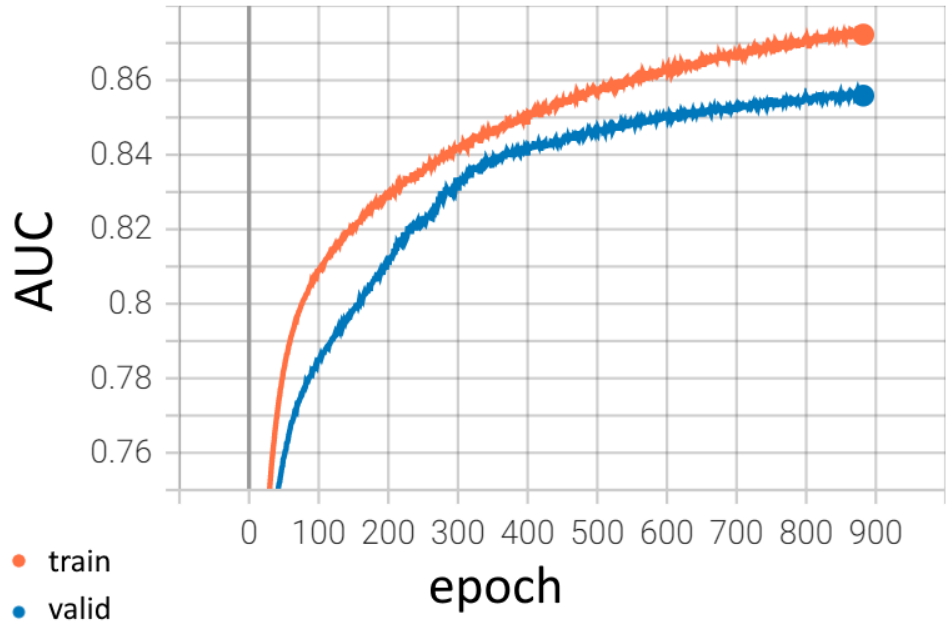

Figure 3.9: Area Under Curve

the label can be a 1 or a 0. The loss function training progress is illustrated in Figure 3.10.

3.6.1 Batch Size

The model training is trained in multiple epochs. An epoch is an iteration over the whole dataset. Each epoch is then subdivided into batches. Finding a batch size that minimalizes the epoch training time is vital to the training process. In our experiments, we have found that a batch size of around 800 000 results in the fastest epoch training time, averaging 6.4 seconds per epoch, in a 10 epoch long training.

3.7 Inference

To create an inference client, we initialize an instance of the InferenceServer-Client class. We pass in the IP address of the container and port 8001 for GRPC inference. Our inference algorithm will infer the results for 50 users at a time, resulting in batches of 1 363 900 predictions per request. One such batch takes around 9.6 seconds to process if we include time spent writing the data to the storage, resulting in a *throughput of approximately 284 000 items*

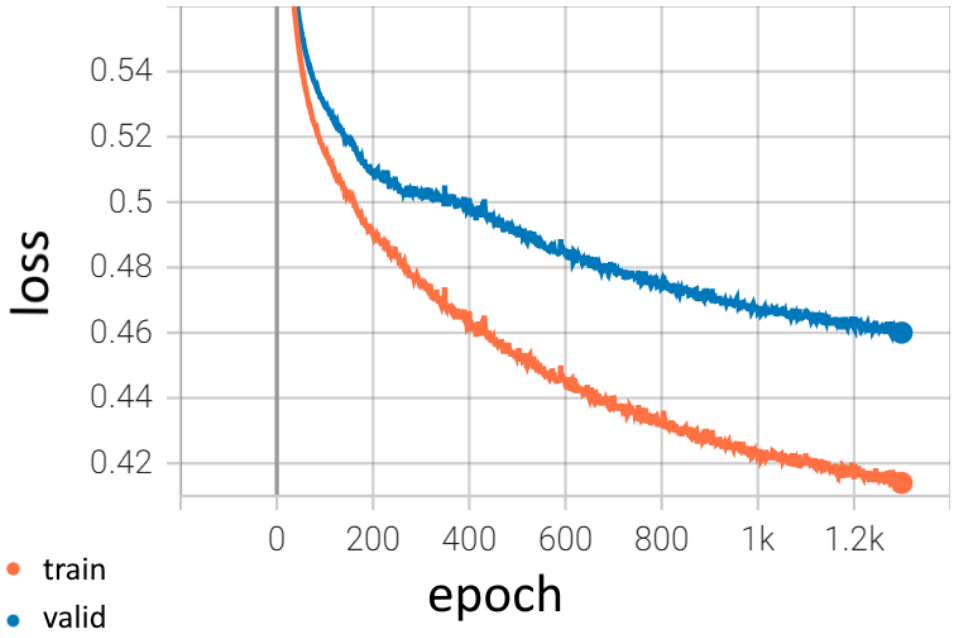

Figure 3.10: Binary Cross-Entropy Loss

per second. Generating the predictions for a single user takes approximately 200 milliseconds.

3.8 Hardware Used

The machine used for the calculation is an NVIDIA DGX with a dedicated single GPU. The GPU model used in this machine is NVIDIA V-100 with 32GBs of HBM2 memory. The total RAM available on the system was 256GB.

Conclusion

In the analysis part of this thesis we have described what is a recommendation system and how the tools provided in NVIDIA Merlin help to implement one. All areas including data preprocessing, model training, and model inference were discussed. The high-level architecture of the Merlin framework was described along with some of the optimalizations NVIDIA managed to implement.

Merlin framework is still undergoing changes that break backwards compatibility and should be considered as a tool under development at the time of writing this thesis. The documentation is not always clear or straight out missing. Exceptions thrown during the development process are usually not clear nor concise and we have had to resort to source code browsing when troubleshooting.

That being said, NVTabular offers a high-level, object oriented and functional approaches to data preprocessing, which enables us to define custom arbitrarily complex operators while maintaining readability. The framework includes good compatibility and integration with TensorFlow.

HugeCTR offers multiple high-level and low-level optimalizations that can help with scaling up the largest recommendation systems.

Triton inference server offers a number of customizations and optimalizations that are crucial when developing a high-performance recommendation system. Statistics are gathered for the loaded models and their lifetime.

A deep learning model was implemented, trained, and used to predict user-item interactions on the MovieLens 20M dataset.

Future work following this thesis is to be supported with this thesis as an introduction to the topic and a guide on how to use Merlin. The following work could focus on implementing state-of-the-art deep learning models, such as Transformers, for commercial purposes.

Bibliography

[1] NVIDIA. Nvidia Merlin Recommender System Framework. Apr 2022. Available from: https://developer.nvidia.com/nvidia-merlin

[2] NVIDIA. GPUDirect storage: A direct path between storage and GPU memory. Jun 2021. Available from: https://developer.nvidia.com/blog/gpudirect-storage/

[3] Floratou, A. Columnar storage formats. *Encyclopedia of Big Data Technologies*, 2018: p. 1–6, doi:10.1007/978-3-319-63962-8_248-1.

[4] Bastani, K.; Asgari, E.; et al. Wide and Deep Learning for Peer-to-Peer Lending. 10 2018.

[5] SiDdhartha. Demonstration of tensorflow feature columns. Oct 2020. Available from: https://medium.com/ml-book/demonstration-of-tensorflow-feature-columns-tf-feature-column-3bfcca4ca5c4

[6] NVIDIA. Triton-inference-server architecture. May 2022. Available from: https://github.com/triton-inference-server/server/blob/main/docs/architecture.md

[7] Spicer, T. Apache parquet: How to be a hero with the open-source columnar data format. Jun 2021. Available from: https://blog.openbridge.com/how-to-be-a-hero-with-powerful-parquet-google-and-amazon-f2ae0f35ee04

[8] Aamir, M.; Bhusry, M. Recommendation system: state of the art approach. *International Journal of Computer Applications*, volume 120, no. 12, 2015.

[9] NVIDIA. Merlin training: Nvidia NGC. Apr 2022. Available from: https://catalog.ngc.nvidia.com/orgs/nvidia/teams/merlin/containers/merlin-training

[10] NVIDIA. Nvidia-Merlin/NVTabular: NVTabular is a feature engineering and preprocessing library for tabular data designed to quickly and easily manipulate terabyte scale datasets used to train deep learning based recommender systems. Jan 2022. Available from: https://github.com/NVIDIA-Merlin/NVTabular

[11] Wijaya, C. Y. 4 categorical encoding concepts to know for Data scientists. Oct 2021. Available from: https://towardsdatascience.com/4-categorical-encoding-concepts-to-know-for-data-scientists-e144851c6383

[12] Glorot, X.; Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[13] Merriam-Webster. Embedding. May 2022. Available from: https://www.merriam-webster.com/dictionary/embedding

[14] Mikolov, T.; Chen, K.; et al. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[15] Cheng, H.-T.; Koc, L.; et al. Wide & Deep Learning for Recommender Systems. *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, 2016, doi:10.1145/2988450.2988454.

[16] Köppen, M. The curse of dimensionality. In *5th online world conference on soft computing in industrial applications (WSC5)*, volume 1, 2000, pp. 4–8.

[17] Giordano, D. 7 tips to choose the best optimizer. Jul 2020. Available from: https://towardsdatascience.com/7-tips-to-choose-the-best-optimizer-47bb9c1219e

[18] NVIDIA. Triton-Inference-server. May 2022. Available from: https://github.com/triton-inference-server/server/blob/r22.04/docs/model_repository.md#model-files

[19] NVIDIA. NGC Overview. Mar 2022. Available from: https://docs.nvidia.com/ngc/ngc-overview/index.html

[20] Harper, F. M.; Konstan, J. A. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.*, volume 5, no. 4, dec 2015, ISSN 2160-6455, doi:10.1145/2827872. Available from: https://doi.org/10.1145/2827872

[21] Liang, D.; Krishnan, R. G.; et al. Variational autoencoders for collaborative filtering. In *Proceedings of the 2018 world wide web conference*, 2018, pp. 689–698.

Appendix A

Acronyms

ETL Extract transform load

API Application programmable interface

GPU Graphical processing unit

NN Neural network

GRPC Google Routing Protocol

Appendix B

Contents of enclosed SD card

- `notebooks` ..... the directory of source JuPyter notebooks
  - `01_Inspect_convert.ipynb` ..... Pandas preprocessing
  - `02_NVTabular.ipynb` ..... NVTabular preprocessing
  - `03_TensorFlow_wide_and_deep.ipynb` ..... Model definition, training
  - `04_inference.ipynb` ..... Inference from Triton server
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format