Master's thesis

NTNU
Norwegian University of Science and Technology
Department of Computer Science

Torsten Bergh Moss

Exploring the Performance of GPU-Accelerated Deep Learning Inference in a Big Data Context with AsterixDB

Master's thesis in Computer Science
Supervisor: Herindrasana Ramampiaro
June 2020

NTNU
Kunnskap for en bedre verden

Torsten Bergh Moss

Exploring the Performance of GPU-Accelerated Deep Learning Inference in a Big Data Context with AsterixDB

Master's thesis in Computer Science
Supervisor: Herindrasana Ramampiaro
June 2020

Norwegian University of Science and Technology
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

To my father Bjørn who let me set up office in his home when the pandemic shut the university down. The man is responsible for sparking my interest in science and technology early, himself being a PhD-level sport scientist, and although his only programming experience was with Simula 67 decades ago he proved an invaluable intellectual sparring partner and rubber duck during our many conversations in the backyard jacuzzi.

To the FlowMotion team of which I am forever grateful to have been a part of. Our experience building a company together developed and shaped me as a person in more ways than you can imagine, and I probably wouldn't be in this field if it weren't for you.

Summary

Motivated by the never ending search for faster and more scalable solutions to process and create value from the vast amount of data generated in our society, we study the performance of GPU-accelerated deep learning inference as a data mining technique for big data in stored and streaming contexts. In order to conduct experiments we chose sentiment analysis of tweets as a task to be performed through deep learning and build a lightweight Recurrent Neural Network with a sparse embedding layer designed to run on the GPU in order to accomplish this. We use AsterixDB as the Big Data Management System providing an environment for persistence and processing of stored and streamed big data, and implement User Defined Functions for AsterixDB using the Recurrent Neural Network to classify tweets. In order to optimize throughput on the GPU, batch processing semantics is implemented in different formats for the User Defined Functions to allow for parallel inference. Our experiments on stored data show linear scalability for increasing dataset sizes with some indications of potential bottlenecks earlier in the processing pipeline, and the streaming data experiments demonstrate a throughput of 75'000 records per second, a tremendous throughput for stream processing and the highest we have seen for this type of experiment with AsterixDB. The results clearly demonstrate the potential of GPU-accelerated deep learning inference as a powerful big data processing technique, however, the nature of our solutions reveals a need for big data management systems and processing engines to better facilitate batch processing semantics in order to easier take advantage of this technology.

Sammendrag

Motiverte av den endeløse jakten på raskere og mer skalerbare løsninger for prosessering og verdiskapning av de enorme mengdene data generert i dagens samfunn studerer vi ytelsen til GPU-akselerert dyp læring som en datagruvedriftsteknikk for lagret og strømmende stordata. For å sette opp eksperimenter velger vi sentimentanalyse av tweets som en oppgave som skal utføres gjennom dyp læring og bygger et lettvekts gjentakende nevralnett med et sparsommelig *embedding layer* designet til å kjøre på GPU. Vi bruker AsterixDB som stordata-håndteringssystem for å skape et miljø for persistens og prosessering av lagrede og strømmende stordata, og implementerer brukerdefinerte funksjoner for AsterixDB som bruker nevralnettet til å klassifisere tweets. For å optimalisere gjennomstrømningen på GPU implementeres semantikk for prosessering av batch i forskjellige formater for de brukerdefinerte funksjonene for å tillate parallell inferens. Våre eksperimenter med lagrede data viser lineær skalerbarhet for økende datasettstørrelser med noen indikasjoner på potensielle flaskehalser tidligere i utførelsesflyten, og strømningsdata-eksperimentene viser en gjennomstrømning på 75'000 tweets per sekund, det høyeste vi har sett for denne typen eksperimenter med AsterixDB. Resultatene demonstrerer tydelig potensialet for GPU-akselerert dyp læring som en kraftig stordata-prosesseringsteknikk, men løsningene våre avslører et også behov for at big data management-systemer og prosesseringsmotorer bedre tilrettelegger for batchprosesserings-semantikk for å enklere utnytte denne teknologien.

Preface

This thesis is written by Torsten Bergh Moss as his final project for the 5 year master's degree program in Computer Science at the Norwegian University of Science and Technology (NTNU). Part of the material related to AsterixDB and GPU processing is based on our preliminary project *GPU-accelerated Multinomial Naive Bayes for Sentiment Analysis of Tweets in a Big Data Streaming Context using AsterixDB* preceding this thesis.

Acknowledgements

This project would not have been possible without my supervisor, professor Heri Ramampiaro at the Department of Computer Science (IDI) at NTNU. His feedback and guidance was an asset of utmost value to me during this research. I also owe a big thank you the AsterixDB community, in particular Ian Maxon at the University of California Irvine, for going above and beyond in helping me with anything and everything that came up when working with their system. Further I would like to thank Paul Dubs of the Deeplearning4j community for the guidance he provided when I was optimizing my deep learning models for the GPU.

Table of Contents

List of Tables

List of Figures

Listings

Abbreviations

CPU	=	Central Processing Unit
GPU	=	Graphics Processing Unit
IoT	=	Internet-of-Things
DM	=	Data Mining
ML	=	Machine Learning
DL	=	Deep Learning
AI	=	Artificial Intelligence
BDMS	=	Big Data Management System
RDBMS	=	Relational Database Management System
UDF	=	User Defined Function
CC	=	Cluster Controller
NC	=	Node Controller
DAG	=	Directed Acyclic Graph
ANN	=	Artificial Neural Network
RNN	=	Recurrent Neural Network
LSTM	=	Long Short-Term Memory
CNN	=	Convolutional Neural Network
TPPS	=	Tweets Processed Per Second
TPU	=	Tensor Processing Unit

Chapter 1

Introduction

This chapter will introduce the thesis' overarching context and concepts, starting with a presentation of the background and motivations behind the research, the research goals and objectives, the approach taken to carry out the research, and the contributions made by the thesis as a result of the research. Finally, the chapter will provide a structural outline for the rest of the thesis.

1.1 Background and Motivation

The lion's share of the technological advancements we have witnessed for the last couple of decades all have in common that they in some way or another spurt a growth in the generation of data. The most obvious ones of these include the emergence of the Web 2.0 (Perrons and Jensen, 2015) which more or less has grown into revolve around the processing and consumption of user generated data, and the introduction of the Internet-of-Things (IoT) (Ashton et al., 2009) connecting us to a world of sensors and embedded systems continuously collecting data. The general digitalization of our society has almost become data-centric. These advancements have given us a world where data is omnipresent, and have given birth to a concept known as Big Data, characterized by the five V's of Volume, Velocity, Variety, Variance and Value (Fan and Bifet, 2013). The first four refer to the nature of the data while the last one is about the focus applied by businesses and organisations to extract and generate value from big data. This is accomplished through processing the data in various ways, more often than not through techniques like Data Mining (DM) and Machine Learning (ML).

While data is being generated at unprecedented rates, our ability to process said data through e.g. DM and ML remains limited by the performance of our hardware, efficiency of our algorithms and the effectiveness and scalability of our systems. The first two often go hand in hand, as certain processing techniques favor different kinds of hardware. Lately, we've seen somewhat of an Artificial Intelligence (AI) boom, characterized by advancements inside the field of Deep Learning, likely due to GPU advancements and the increased use of GPU's to handle the underlying algebraic computations for deep learn-

ing. This thesis will give special attention to the performance and use of GPU-accelerated Deep Learning in order to mine big data.

Due to the nature of big data, there is a need to be able to perform the processing on both stored static data and streaming data. There exists a myriad of tools and systems for both storing, streaming and processing big data, often leading to final architectures of ”glued together” systems. AsterixDB[1] is a Big Data Management System (BDMS) built on the concept of ”one size fits a bunch” with the goal of removing the need to develop these glued systems and therefore allow organisations to spend less time developing the system and more time performing the data analysis. Out of the box AsterixDB provides both data persistence, query and processing capabilities for persisted data, and a way to handle and process data streams, making it a natural choice for exploring the performance of GPU-accelerated deep learning inference on different shapes and forms of big data. We find additional motivation in the fact that delegating the process of data mining to specialized hardware such as a GPU, a cluster's CPU resources can focus on database specific tasks of reads/writes and answering queries, making for a more effective system in total.

1.2 Goals and research questions

The main goal of the thesis is to investigate the performance in terms of speed and scalability of GPU-accelerated deep learning inference bundled as a User Defined Function (UDF) in a BDMS like AsterixDB, as a data mining technique for big data. To investigate this we chose *sentiment analysis of tweets* as the data mining task to be carried out using deep learning inference, reasons being that this is a task with a wide range of practical applications, as well as being a field frequently researched within the context of AsterixDB (Pääkkönen, 2016) (Abrahamsen, 2017) (Alkowaileet et al., 2018) (Finckenhagen, 2018) (Moss, 2019), giving the UDF's developed and the results obtained in this study a broad body of research for comparison. Formulated more concretely as a main question for which we will seek an answer, we ask:

> *How does GPU-accelerated deep learning inference scale as a data mining technique for big data?*

On our way to answer this research question we will investigate the most most efficient and effective deep learning techniques for sentiment analysis of tweets, as well as the optimal processing semantics for GPU-based deep learning inference on streamed and stored data and how they can be realized in AsterixDB's UDF framework in order to scale to big data.

[1] http://asterixdb.apache.org/

1.3 Approach

1.3.1 Research method

The research in this thesis started with a literature review on deep learning techniques for sentiment analysis, GPU-accelerated deep learning, machine learning techniques for big data, as well as literature on AsterixDB, in order to identify the most effective approach to tackle the problem.

Then based on the information gained from the literature review we performed development work creating a neural network for sentiment analysis and developing AsterixDB UDF's utilizing this network to classify tweets as well as realizing optimal processing semantics for maximum throughput.

Finally we performed experiments for these UDF's scaling the volume and velocity of the big data they had to process, before a discussion of our results featuring a comparative analysis with other similar experiments done on the AsterixDB UDF framework.

1.3.2 Scope and limitations

We find it imperative to specify that although the research will focus on sentiment analysis of tweets, the task in particular is not the main scope of the thesis, neither is achieving a particularly high accuracy for it. The main scope of the thesis is to investigate how deep learning on the GPU scales in terms of volume and velocity of stored and streamed big data, and sentiment analysis is merely used as a task to be carried out by deep learning to investigate this.

When conducting the research done in this thesis there were limitations for cluster scaling. Previous work with AsterixDB has simulated a scaling cluster by using more CPU-cores as cluster nodes, however, because the UDF in this thesis does the main processing work on the GPU, simulating a scaling cluster the same way would make no sense as the UDF's computational resources technically aren't scaling. To explore the same kind of scalability of a system as the one developed in this thesis we would have to acquire a large cluster of machines comprising of both CPU's and GPU's, e.g. by cooperating with a large technology company with access to that sort of hardware.

AsterixDB and it's UDF framework is written in java, therefore the implementation of the UDF necessarily has to be written in java also. For deep learning, the most popular frameworks are python based.

1.4 Contributions

1.4.1 Main contribution

The first contribution of this thesis will be a lightweight Recurrent Neural Network for sentiment analysis built with speed and scalability in mind. The main contribution of this thesis will be the UDF's implementing this neural network in a way that realizes GPU-accelerated deep learning parallel inference on stored and streamed data, along with a presentation of how this is accomplished. Further, the thesis contributes an experimental study of how these UDF's scale with increasing velocity and volume of data.

1.4.2 Significance

The results of study can help serve as a guide and inspiration for choosing hardware, tools and algorithms for performing large scale machine learning tasks both on stored data and on datastreams. One can envision e.g. a company making a tool predicting the future value of crypto currencies wanting to use sentiment analysis of real time social media data as part of that analysis, and therefore consider using a mixed hardware cluster of CPU's and GPU's performing sentiment analysis on a realtime twitter stream through deep learning inference, such as researched in this work. Also, because most *de facto* tools for Big Data and ML are specialized to only solve part of the problem, they have to be glued together into complex architectures in order to provide a final system for both stream processing, stored data processing and persistence. Therefore, a significant value of the contribution of this thesis will be showing that a generalised out-of-the-box "one size fits a bunch" system like AsterixDB with considerably less moving parts and development work can perform at a higher level than more complex architectures of specialised state-of-the-art tools.

All of this ultimately could lead to a paradigm shift for the way we chose to process and store big data in the future.

1.5 Thesis structure

Chapter 2 will thoroughly present the background theory for the main technological concepts of the thesis. In two subsections it will present the the workings of AsterixDB from a user's perspective as well as the mechanics of how it works under the hood. Further, it will go into depth presenting the concept of deep learning and the mechanics of the deep learning techniques used in this thesis, as well as present theory behind GPU computing and using GPU's for deep learning. Chapter 3 will present relevant related work on big data, GPU-based machine learning and deep learning. Chapter 4 will present the contributions of this paper, a lightweight recurrent neural network designed perform sentiment analysis efficiently and effectively on the GPU, as well as a description of how this network is implemented into AsterixDB UDF's to realize optimal processing semantics. Chapter 5 will showcase experiments used to test the performance of the UDF's presented in Chapter 4. Chapter 6 will discuss the results of the experiments in Chapter 5 and compare them to results achieved in similar research. Chapter 7 will conclude this thesis and present suggestions for future work.

Chapter 2

Background theory

This chapter will elaborate on the different background theory necessary to understand the work completed in this thesis. Section 2.1 will provide a thorough introduction to the AsterixDB system. Section 2.2 will provide a theoretical introduction to the concepts and mathematics of deep learning. Section 2.3 will cover relevant background theory on GPU's and GPU programming.

2.1 AsterixDB

This thesis is preceded by a preliminary project investigating the performance of a GPU-based implementation of Naive Bayes algorithm with AsterixDB (Moss, 2019). The preliminary project thoroughly covered how to use AsterixDB and the overlying concepts. The presentation of AsterixDB from section *2.1 AsterixDB* in the preliminary project is included below under section 2.1.1 AsterixDB over the hood, to provide a view of the most important AsterixDB concepts from a user's or data scientist's perspective as well as a short how-to introduction. The section 2.1.2 AsterixDB under the hood gives special attention to the underlying mechanics of the AsterixDB system in order to provide the necessary background for the work done realizing optimal processing semantics through AsterixDB's UDF framework.

2.1.1 AsterixDB over the hood

The AsterixDB project began in 2009 at the University of California Irvine, went open-source in 2013, and is today an open-source BDMS in full bloom. It draws natural characteristics from both Data-Warehouses and Analytical Engines, making it suitable to both store, manage and analyse Big Data. (Alsubaiee et al., 2014) Some of the key features includes

- A flexible, semistructured data model allowing the user to control the degree of

schema-validation. It includes support for all of today's "Big Data"-types like textural, temporal, and spatial data values.

- A full-fledged expressive query-language SQL++ that has a parallel query runtime. It is a superset of normal SQL, and adapted to work with AsterixDB's data model.
- Support and framework for customizable UDF's written in Java or SQL++, and the ability for the user to easily build and deploy them.
- Support for continuous data ingestion in streams, and support for the continuous analysis of this ingested data using UDF's attached to the streams.

The following sub-sections provides a logically structured and deeper introduction to some important concepts and key terminology serving as background to understand the work done with AsterixDB in this thesis.

Dataverses, Datasets and Datatypes

While we in Relational Database Management Systems (RDBMS) are used to working with the concepts of Databases and Tables, the corresponding concepts for AsterixDB are called *Dataverses* (short for data-universes) and *Datasets*, with a Dataverse being the top-level organizing concept. Inside of this Dataverse one has the ability to define own Datasets, functions, artifacts and Datatypes. Utilizing a semi-structured data-model, the concept of "Closed vs Open" Datatypes is introduced, *Closed* meaning the instance will have to follow some schema rules, and *Open* providing the opportunity to add additional content, thus providing figurative "wiggle room". The data in the Datasets are using AsterixDB's own customized flexible JSON-based data model called ADM. Architectually speaking, ADM is a superset of JSON, and a natural product of adding more data types and data modelling constructs to JSON. (Alsubaiee et al., 2014)

Listing 2.1 illustrates the use of SQL++ to define a Dataverse *TweetSentimentAnalysis*, Dataset *UnprocessedTweets* and Datatype *Tweet*, as we will see them being used in examples ahead.

```
1  -- Create and use a Dataverse
2  DROP DATAVERSE TweetSentimentAnalysis IF EXISTS;
3  CREATE DATAVERSE TweetSentimentAnalysis;
4  USE TweetSentimentAnalysis;
5
6  -- Create an Open Datatype to serve as a model Tweets
7  CREATE TYPE Tweet AS OPEN {
8    id : int64 ,
9    text : string
10 };
11
12 -- Create a Dataset to hold data of type Tweet
13 CREATE DATASET UnprocessedTweets ( Tweet );
```

Listing 2.1: Demonstrating the creation of Dataverse, Datatype and Dataset

Data Feed Creation

AsterixDB achieves continuous data ingestion through it's data feed-mechanism, which allows for incremental population of a Dataset as data is streaming into the feed. Data Feeds are brought into life using *feed adaptors*, which are implementations of an interface with details specific to the data source. The adaptor will function in either *Push*- or *Pull*-mode, with the former just involving a single request/handshake from the feed before the data is continuously "pushed" into the adaptor, and the latter having the adaptor send requests at time intervals to the source before receiving the newest data in a micro-batch fashion. The user can choose to configure their own adaptors to listen to a socket, RSS, or it can use one of AsterixDB's built in adapters, like the Twitter adapter which is built on the Twitter Streaming API. (Grover and Carey, 2014)

Listing 2.2 shows the creation of a Twitter-Feed operating with push-mechanics. It assumes the user already has a developer account with Twitter, and thus access to authentication-parameters like *consumer.key*, *consumer.secret*, *access.token* and *access.token.secret*. The life-cycle of a feed follows a pattern of Creation-Connection-Start-(Stop)-(Disconnect), which is demonstrated in the listing, with line 5-15 being the creation and definition of the feed, line 18 connecting the feed to a Dataset and line 19 starting the feed so data can be ingested. The feed can be disconnected from the Dataset, or stopped altogether, using the `DISCONNECT FEED` or `STOP FEED` operators.

```
1  -- Enters Dataverse
2  USE TweetSentimentAnalysis;
3
4  -- Create a Twitter Feed using built-in adaptor
5  CREATE FEED TwitterFeed WITH {
6    "adapter-name": "push_twitter",
7    "type-name": "Tweet",
8    "format": "twitter-status",
9    "language": "en",
10   "consumer.key": "**********",
11   "consumer.secret": "**********",
12   "access.token": "**********",
13   "access.token.secret": "**********",
14   "keywords": "[Your keywords here]"
15 };
16
17 -- Connect feed to Dataset and starts
18 CONNECT FEED TwitterFeed TO DATASET UnprocessedTweets;
19 START FEED TwitterFeed;
```

Listing 2.2: Creating a Twitter-Feed in AsterixDB

User Defined Functions

From RDBMS we know the concept of a *view* and how it can be used to transform and aggregate data into a new processed set that can be queried, however, sometimes one will want to perform more advanced processing outside of the scope of the system's query-language. AsterixDB solves this problem by introducing the concept of External User Defined Functions, allowing the user to define it's own functions using a programming language like Java, package it and deploy it to a Dataverse. The types of processing achievable

are thus only limited by the imagination, as the user more or less could write any program including any external library, bundle it, deploy it, and have it interacting with the data inside of AsterixDB.

The UDF follows a factory-function-pattern, similar to those seen inside the Domain Driven Design methodology, where the Factory is responsible for creating an instance of the Function at runtime. The Function's lifecycle includes an initialization-step that catalyzes the UDF, an evaluation step that does the actual processing, and a de-initialization-step where the user can do any cleanup-operations necessary e.g. to prevent memory leaks. Once both the Factory and Function is populated with the desired logic one has to create an XML library configuration file containing the metadata necessary for AsterixDB to start using the UDF. As we can see from the example of a hypothetical Sentiment Analysis UDF in listing 2.3, this information includes defining the name of the function, the type, input and return values and Factory-definitions. (Alkowaileet et al., 2018)

```
<libraryFunctions>
    <function_type> SCALAR </function_type>
    <name> classifyTweet </name>
    <arguments> Tweet </arguments>
    <return_type> Tweet </return_type>
    <definition> org.apache.asterix.external.library.
        SentimentFactory
    </definition>
</libraryFunction>
```

Listing 2.3: A metadata function description for a Sentiment Analysis UDF

After the UDF is packaged, using e.g. Maven in the case of a Java-UDF, it can be deployed to a Dataverse sending a POST-request to the URL of the Dataverse's Library-directory with a binary body containing a compressed folder with the UDF-files. Building on previous examples, listing 2.4 shows how one can run the UDF in a query-fashion on all entries in a Dataset. Listing 2.5 shows how the same UDF can interact with a stream of Tweets entering a Dataset, performing processing and giving the Tweets a sentiment even before they even enter the Dataset.

```
-- Enter Dataverse
USE TweetSentimentAnalysis;

-- Run UDF on all tweets in UnprocessedTweets
SELECT library#classifyTweet(t) FROM UnprocessedTweets AS t;
```

Listing 2.4: Calling a UDF on a dataset

```
1  -- Enter Dataverse
2  USE TweetSentimentAnalysis;
3
4  -- Create a new Dataset to put the pre-processed Tweets
5  CREATE DATASET ProcessedTweets(Tweet) primary key id;
6
7  -- Connect existing feed to Dataset and apply UDF
8  CONNECT FEED TwitterFeed TO DATASET ProcessedTweets APPLY FUNCTION library
       #classifyTweet;
9
10 -- Start feed
11 START FEED TwitterFeed;
```

Listing 2.5: Using a UDF in a streaming context

2.1.2 AsterixDB under the hood

The AsterixDB system carries out the features and functionality presented in 2.1.1 using a multitude of software components in a finely composed software stack realizing the system architecture. The most important ones, and of which we will elaborate on below, is *Hyracks* which is used as a scalable parallel runtime execution engine, and *Algebricks* which is used for optimization and implementation of the SQL++ query language. A high level FMC diagram (Keller and Wendt, 2003) of the system can be seen in figure 2.1. It depicts the structure of the AsterixDB cluster with the Cluster Controller (CC) acting as a master node of the cluster. The CC communicates with the different clients of the outside world, compiles queries, and coordinates the work performed by the Node Controllers (NC) which function as the worker nodes in cluster. The AsterixDB software stack can be seen in figure 2.2. The figures and information in this section is based on the introductory paper on the AsterixDB system by Alsubaiee et al. (2014).

Hyracks

The most bottom layer of the AsterixDB software stack is the Hyracks runtime execution layer which is responsible for receiving and administering computation jobs in parallel, requested by the layers above. These computation jobs are represented as Directed Acyclic Graphs (DAG) which are made up of *operators* and *connectors*, where operators are components consuming input partitions and producing output partitions, and connectors redistribute data from the output partitions and supply input partitions for the DAG's next Operator. An Operator usually include one or two *activities*, and during execution all operators are expanded into their activities in order to identify which activities provide blocking requirements and which can be executed in parallel. The original Hyracks paper by Borkar et al. (2011) saw Hyracks outperforming state-of-the-art Hadoop for a number of use cases.

Algebricks

Algebricks describes itself as a data-model agnostic compiler backend for big data languages (Borkar et al., 2015), and is used as backend for optimizing the SQL++ queries a

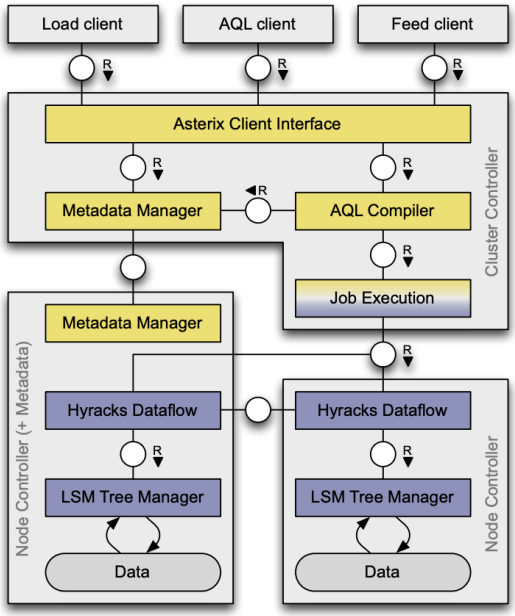


Figure 2.1: The AsterixDB System Architecture as presented by Alsubaiee et al. (2014). We note that since their paper was published, AQL has been deprecated and succeeded by SQL++.

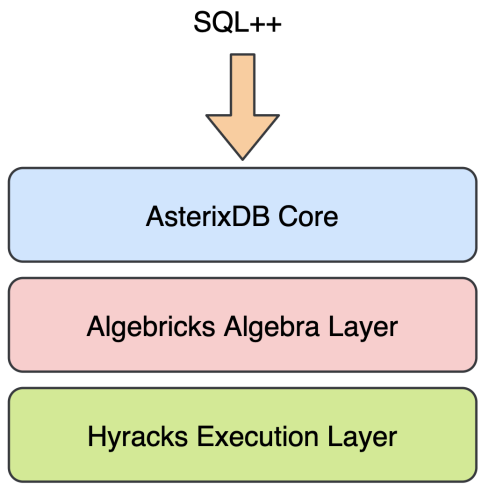


Figure 2.2: The AsterixDB software stack, also adopted from Alsubaiee et al. (2014). The figure is cropped to only include the AsterixDB parts of the original Asterix stack, which also includes support for other query languages and software components, as well as ports for Hive and Hadoop.

user might perform through AsterixDB. The Algebricks layer can be seen just above the Hyracks execution layer in the software stack of figure 2.2. When AsterixDB receives an SQL++ query it is compiled down to an Algebricks algebraic program which is then re-written using algebraic rules in order to introduce partitioned parallelism for scalable execution. After this optimization the resulting query execution plan is translated into an Hyracks job and executed on the Hyracks execution layer just below it.

Data Feed Adapter and Ingestion Pipeline

Once a feed is started in AsterixDB such as demonstrated in listings 2.2 and 2.5, the SQL++ query will be compiled down to a Hyracks job that will be due for continual execution for as long as the feed is alive. We call this Hyracks job a *feed ingestion pipeline* and as any Hyracks job it mainly consists of operators and connectors. As can be seen in figure 2.3, the operators and connectors used are the *intake*, *compute* and *store* operators along with the data connectors. As we can see, the first operator of any feed ingestion pipeline is the intake operator which as the name implies is responsible for taking in data from the stream and converting it to records fitting AsterixDB's data model. Next, it sends the records to a data connector which distributes them to the compute operators which are responsible for applying any UDF's and computing it's results. Finally the data is sent through another layer of data connectors into the store operator which stores the result of the processed datastream to an AsterixDB dataset. The cluster's CC node hosts a *Central Feed Manager* which is in charge of scheduling execution of the active pipelines by assigning operators to NC nodes and manage the degree of parallelism of the intake and compute operators, while the store operator has it's parallelism predetermined by the nodes that hold the dataset partitions. In the case that the datastream speeds up and more resources is added to the cluster, the pipeline might be restructured by the central feed manager, depending on the ingestion policy (Grover and Carey, 2014).

2.2 Deep learning

Deep learning is a subset of the machine learning field, focusing on the use of Artificial Neural Networks (ANN) mimicking the way the human brain works in order to conduct supervised and unsupervised learning. There is some discussion as to exactly when the concept of deep learning and neural networks originated, as it has been a gradual development of concepts with the very first references of computational neurons dating all the way back to 1943. However, the first algorithmic implementations of a supervised learning multilayer neural network was first introduced by Ivakhnenko and Lapa in 1965. (Schmidhuber, 2015) Since then the concept gained traction in short scattered bursts, but it didn't break through to the "mainstream" before the 2010s when paired with Nvidia GPU's the overall processing speed skyrocketed making deep learning a very feasible ML technique. Since then it has been a go-to technology for many ML tasks.

This section will give a brief introduction to the mechanics of neural networks in general as well as the specific workings of the class of neural networks known as recurrent neural networks. Further it will go into detail on techniques used to have neural networks classify text.

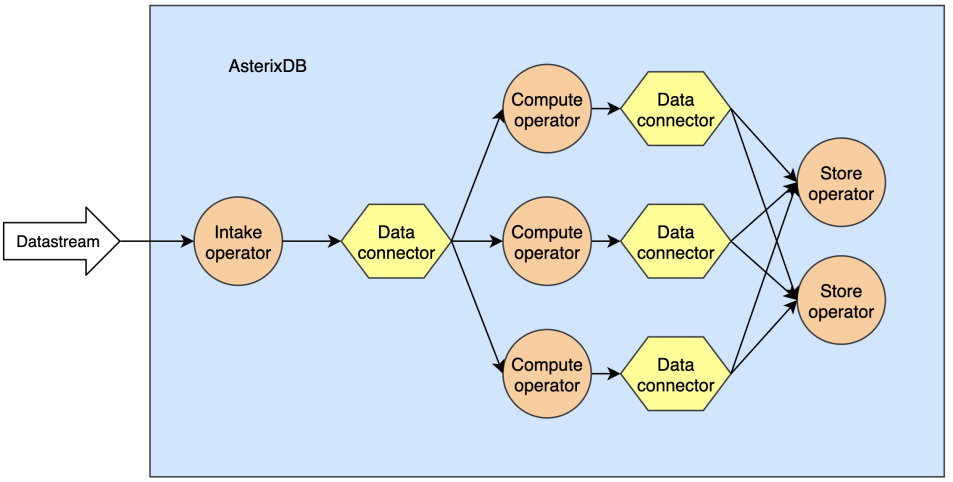


Figure 2.3: A feed ingestion pipeline in AsterixDB.

2.2.1 Neural Networks mechanics

Artificial neural networks are computational structures used for inference as way to perform machine learning tasks. The structures vaguely mimic the way biological neurons work in the human brain, hence the name. Simply put, the network consists of processing nodes, ”neurons”, and the weighted connections between them, arranged in a layer-like fashion with an input layer, a variable number of hidden layers and an output layer, each containing a number of neurons. The neurons all have some kind of activation function that may produce an output upon receiving the appropriate signals from nodes in the previous layer. A simple neural network with an input layer of three neurons, a single hidden layer of three neurons, and an output layer of two neurons can be seen in figure 2.4.

While for a lot of supervised machine learning techniques the processes of training the model and using it for inference are independent from each other and different in nature, for neural networks the two share some procedures. ANN's follow a training pattern of performing the inference, looking at the results produced and tuning it's parameters appropriately. In other words, an ANN in training mode will perform inference based on training data, compare the results to ground truth and use the potential errors to update the network's parameters, while an ANN in inference mode simply will perform the inference and be done. Because the process of inference is the same in both cases this chapter will first present the most important concepts related to it, particularly *forward passing* and *activation functions*, then move on the most important concepts related to what happens after inference in the training case, namely *loss functions* and *backpropagation*.

Forward pass

As the name might suggest, forward passing is the process of passing values through the network by feeding the input layers with input values, which typically for ML tasks will

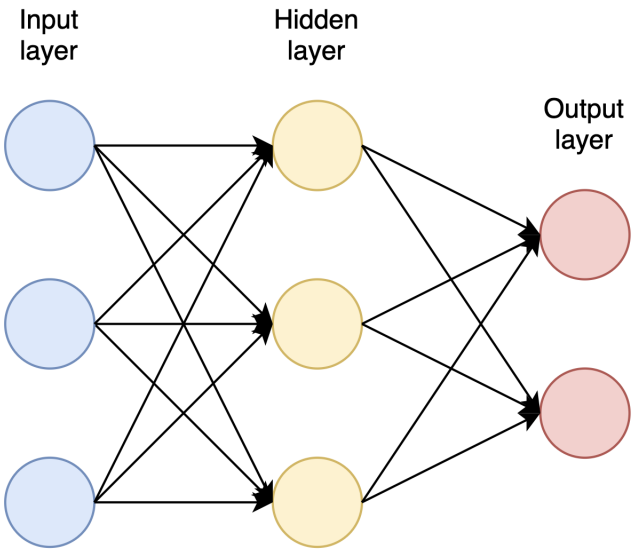


Figure 2.4: A simple neural network demonstrating the different types of layers.

be some sort of feature vector, and letting the neural activations propagate signals through the network until it arrives in the output nodes as one or more output values. Figure 2.5 attempts to showcase the mechanics of a single hidden neuron in a more mathematical context. We write the input from a single neuron in the previous layer as $x_i w_i$ where x_i refers to the value of the output of the activation function from that neuron, and w_i is the weight of that particular edge. From this we define the input z to this neurons activation function as

$$z = x_1 w_1 + x_2 w_2 + ... + x_i w_i + b = \boldsymbol{w}^T \cdot \boldsymbol{x} + b \tag{2.1}$$

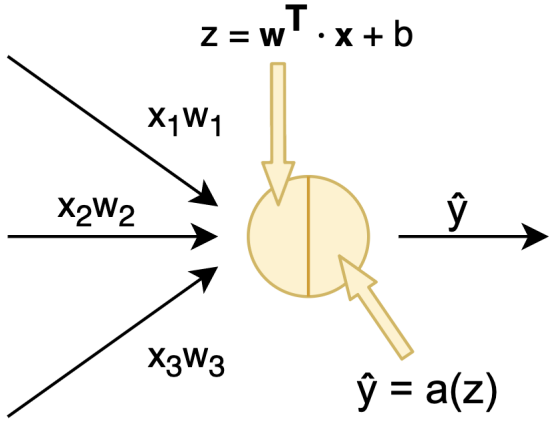
where the weights and x-values respectively are put into a column vector and row vector, and a bias b is introduced as an additional model parameter. The output of the neuron $\hat{y}$ is determined by

$$\hat{y} = a(z) \tag{2.2}$$

where $a(z)$ is the neurons activation function. Like this the neurons in the net receive, compute and feed forward values until the net has produced one or more outputs to serve as basis for inference.

Activation functions

Activation functions represents the amount of activation of a single neuron and produces the neuron's output value to be fed forward to the next layer in the net, or in the case that it's an output neuron, serve as basis for the inference. Some of the most widely used



.

Figure 2.5: One of the neurons in the hidden layer of figure 2.4. We visualize the first half of the neuron being responsible for computing the effect z of the input edges and their activations, and the other half of the neuron for computing it's own activation $\hat{y}$

activation functions, both in this thesis and across the board, can be seen in figure 2.6 plotted in a cartesian coordinate system. Figure 2.6a shows the **ReLU** activation function characterized by the equation

$$a(x) = \begin{cases} \text{x}, & \text{if } 0 < x \\ 0, & \text{otherwise.} \end{cases}$$

Figure 2.6b shows the **Sigmoid** activation function characterized by the equation

$$a(x) = \frac{e^x}{e^x + 1}. \tag{2.3}$$

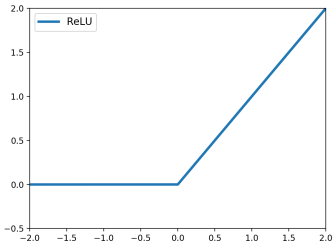
Figure 2.6c shows the **Tanh** activation function characterized by the equation

$$a(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.4}$$

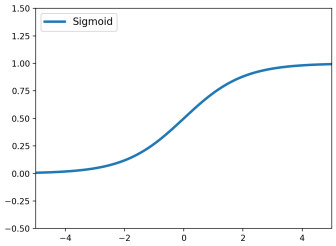
The last activation function that has to be mentioned is the **Softmax** activation function characterized by the equation

$$a(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}}, \tag{2.5}$$

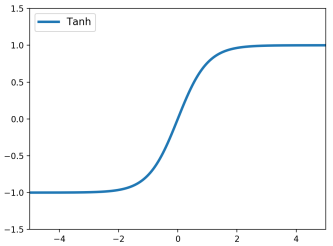
which is slightly different from the other's as rather than produce a single activation value, it is meant to produce a probability distribution across output neurons as a way to do multi-label classification. It takes an input vector $\boldsymbol{x}$ of values from the previous nodes and uses



(a) The ReLU activation function



(b) The Sigmoid activation function



(c) The Tanh activation function

Figure 2.6: Plots of some widely used activation function

them to cast the probabilities of the input belonging to the different possible output classes, where $a(x)_i$ is the probability predicted by the net that the input has class i.

Loss functions

In the case of training the ANN, after a full forward pass has been completed, the net will use a loss function as a way to measure how wrong the output is from ground truth or the ideal solution. The goal when training the ANN therefore will be to find a way to minimize the computed loss, and as the training proceeds ideally one would want to see an inverse relationship between the nets accuracy and loss, where as the accuracy gets higher and higher the loss decreases accordingly.

Loss functions are chosen based on the type of inference done by the neural network as well as the activation functions used by the output layer. Typically for regression tasks the **Mean Squared Error** loss function characterized by the equation

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

is used, where y_i is the ground truth or ideal value at regression step i, and $\hat{y}_i$ is the ANN's predicted value at the same step. For inference tasks dealing with classification the output neurons typically use activation functions that produce some probability distribution over the different possible classes. To calculate loss between the predicted distributions and the

ideal ones we typically use the cross entropy measure. For multi-class classification one typically uses **Multi Class Cross Entropy**, often referred to as Categorical Cross Entropy or simply Logarithmic loss. This type of loss function is normally seen with an activation function such as the softmax described by equation 2.5. The general cross entropy loss function is characterized by the equation

$$CE = -\sum_{i}^{C} y_i log(\hat{y}_i) \tag{2.6}$$

where $\hat{y}_i$ and y_i are the predicted values and ground truth for the class i. For the case that there are only two classes, $C' = 2$, one often refers to the **Binary Cross Entropy** loss function which is a special case of the equation 2.6 characterized by

$$CE = -\sum_{i=1}^{C'=2} y_i log(\hat{y}_i) = -y_1 log(\hat{y}_1) - (1 - y_1) log(1 - \hat{y}_1) \tag{2.7}$$

where the function has been manipulated in such a way that technically only the class y_1 is present, as the nature of probability implicitly also describes the probabilities of y_2 since the probabilities have to sum to 1. This reduces the need for output neuron from two down to one, as a high activation in the single output neuron would indicate a high probability of one class and an equivalent low probability of the other class.

Backpropagation and optimizers

After a forward pass has been done and loss has been calculated the ANN uses the process of backpropagation to tune the edge weight and bias parameters in a way that would have reduced the value of the loss function and in that way ”learn” how to classify the inputs correctly. This is accomplished through different *optimizers* which are algorithms or techniques used to update the parameters. Most of these algorithms and techniques have grown effectiveness, but also in complexity both algorithmically and mathematically, as the field has progressed over the years, and describing them in their entirety would be outside of the scope of this thesis. However, most of them stem from one of the most famous optimizers known as **Gradient Decent**, where the general idea is to compute the gradient of the loss function with respect to the parameters in order to determine which way to adjust the parameters in order to decrease the loss value. This idea can be described mathematically by

$$\boldsymbol{p}_{n+1}^{[l]} = \boldsymbol{p}_{n}^{[l]} - \alpha \nabla F(\boldsymbol{p}_{n}^{[l]}) \tag{2.8}$$

where the vector of parameters $\boldsymbol{p}$ at layer $[l]$ and gradient descent step $n+1$ is updated to be it's values at the previous step n moved amount $\alpha \in \mathbb{R}_+$ in the negative direction of the gradient of the loss function F with respect to the parameters in $\boldsymbol{p}$. Here α is known as the learning rate, and if it is sufficiently small it naturally follows that $F(\boldsymbol{p}_{n+1}^{[l]}) \leq F(\boldsymbol{p}_{n}^{[l]})$, which is the behaviour that we want as it means loss is decreasing and therefore accuracy increasing. The parameters we have available for tuning are typically the weights and biases at each layer. Beginning at the output layer, as the weights and biases are updated

according to 2.8, the algorithm also keeps track of how the activation values of the previous layer would have to change in order to minimize the loss function, as they too influence the output values even though they technically are not parameters. With this in mind the algorithm recursively computes the gradient descent for each preceding layer, propagating backwards until the input layer is reached. This process of beginning with the output in mind and backwards updating the parameters is what is known as **Backpropagation**.

One of the most popular optimizers, which also is widely used in this thesis, is known as the Adam optimizer (Kingma and Ba, 2014). Adam is a variant of gradient descent that has it's name from it's use of adaptive moment estimation, using the first and second moment of the gradient to tailor the learning rate α for each parameter in the network, thus finding the global minimum of the loss function faster during training.

Epochs

Using the techniques described above one can clearly see how the ANN learns as it is fed training data that it uses to fit it's parameters. Unlike some other machine learning techniques, such as i.e. Naive Bayes, an ANN can keep learning and improve performance by seeing the same training data multiple times. One full pass of all the training data available is known as an **Epoch** and the amount of epochs required to see optimal results in an inference setting will vary both with the amount of training data available as well as with the nature of the inference task at hand.

2.2.2 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a special type of neural network that aims to instill a certain element of memory in the network, where given a sequence of inputs it will consider the previous elements in the sequence when performing inference for the current element. This mimics in some way how humans process e.g. textual information, making it a great fit for natural language processing tasks like sentiment analysis. Looking at a sentence like "*I am happy, not sad.*" we immediately see the advantage of an approach like this compared to other ML techniques like e.g. Naive Bayes that relies on assuming each word to be independent, as the positions of the words "happy", "sad" and "not" are crucial in determining the sentimental value of this sentence.

Figure 2.7 depicts a simple RNN, with the left hand side of the equation being an architectural overview over the network and the right hand side being the same network "unraveled" over a temporal axis depicting the network's behaviour from timestep 1 to n. To allow for the illustration of a temporal axis when creating an RNN-figure we imagine that we are seeing the net "from birds perspective" and that below the neurons in each layer there could be number of additional neurons. Therefore, in the succeeding figures what looks like a single neuron is an entire layer and the arrows display the flow of a vector through the network rather than a single number. The right hand side of the equation in the figure depicts input vectors $x_1, x_2, ..., x_n$ that are fed in sequence to the network. For each input the network produces or updates a hidden state $h_1, h_2, ..., h_n$ that is passed along as an additional input at the next step in the sequence. To provide some intuition one can envision this to be a simple RNN performing sentiment analysis and that the input vectors $x_1, x_2, ..., x_n$ each are some vectorial representation of words in a sentence that should

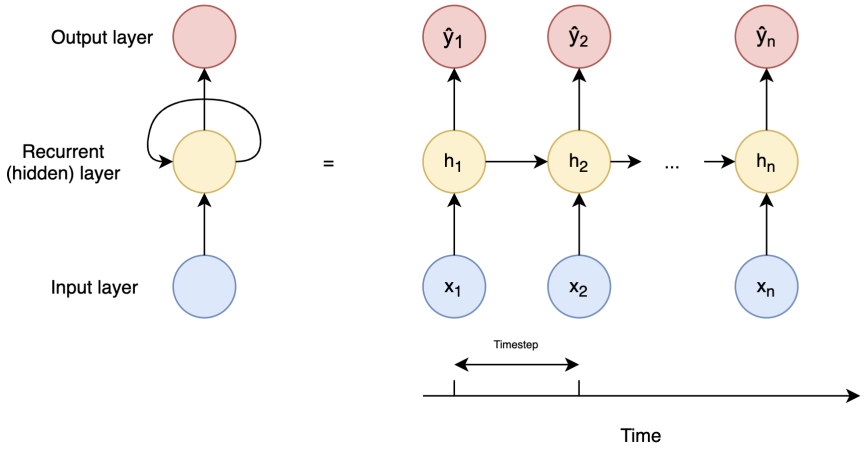


Figure 2.7: A simple Recurrent Neural Network

be classified as negative or positive. At each iteration the network will "remember" and account for previous inputs through the hidden state, and produce a predicted sentiment $\hat{y}$ based on the inputs that it has seen so far. The predicted sentiment of the whole sentence will therefore naturally be the last output, after all words are processed, $\hat{y}_n$.

Long Short-Term Memory

A significant problem that emerges during the training of RNN's is what has come to be known as the Vanishing Gradient Problem. The problem was first described by Hochreiter (1991) in his master's thesis written in German, then later expanded on by Hochreiter et al. (2001) in an English paper. It arises during backpropagation when the RNN trains on long input sequences and the gradient for the loss function is calculated "back in time" across the timesteps of the recurrent layer in order to update the weights. As the product of the gradients at each timestep propagates backwards it will tend to become diminishingly small before the whole backpropagation is complete, often resulting in the RNN being unable to learn at all as equation 2.8 reveals that the parameters will remain virtually unchanged.

As a solution to the vanishing gradient problem Hochreiter and Schmidhuber (1997) created the Long Short-Term Memory (LSTM) network architecture which can be seen in figure 2.8. The reason we label LSTM as a network architecture rather than a neuron is the fact that it actually self consists of multiple neurons and operators. In the figure, at timestep t the input vector x_t and the output vector $\hat{y}_t$ are depicted in the same style as previous figures. The arrows in the figure are showing the flow of a vector of numbers, and when a given path branches of the vector is copied in order to follow both paths. When two paths cross the vectors from each path are concatenated. The figure's purple boxes are each meant to represent a neural network layer, with the σ referring to a layer with neurons using the sigmoid activation function of equation 2.3 and $tanh$ referring to a layer

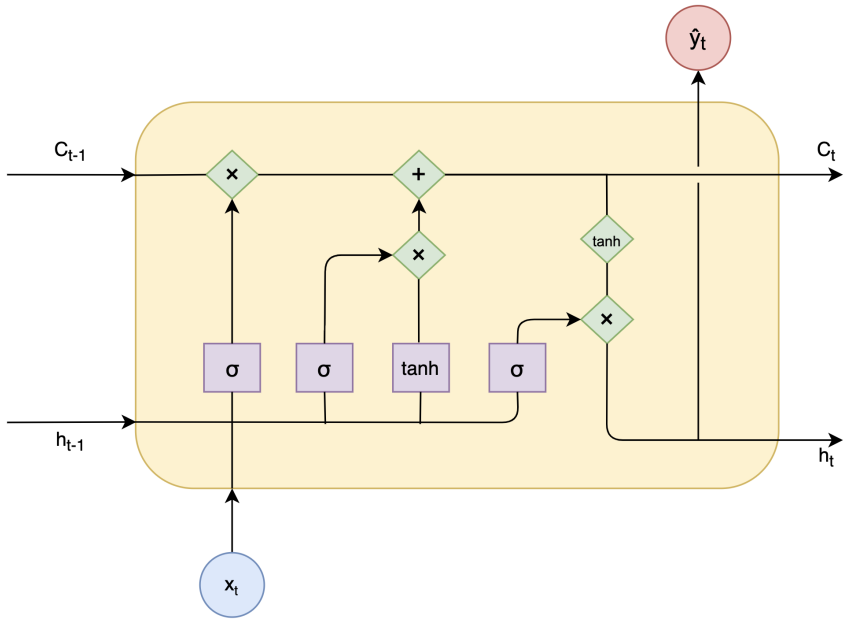


Figure 2.8: The LSTM architecture.

with neurons using the hyperbolic tangent activation function of equation 2.4. The green tilted rectangles are pointwise addition, multiplication or hyperbolic tangent operators. As can clearly be seen from figure 2.6b the sigmoid function outputs values between 0 and 1, and in combination with the pointwise multiplication operator they constitute a filtering gate controlling how much of the signal, mathematically the length of the vector, will be let through. This gate is seen three places in the LSTM architecture and is used to perform state control. The lower horizontal path denoted by a subscripted h carries the output at the previous timestep and sends out the output at the current timestep. The upper horizontal path known as the *memory cell* and denoted by a subscripted C is one of the main features of the LSTM network and works like a conveyor belt passing some information through the network. The output from the previous timestep h_{t-1} together with input x_t control through the leftmost gate how much of the information on the memory cell should be forgotten, and through the sigmoid and hyperbolic tangent layers to the right how much information from the current inputs should be added to the cell. The output then is decided by sending the information on the memory cell through a pointwise hyperbolic tangent operator and a another gate controlled by the input and previous output. The memory cell is an essential part of how LSTM solves the vanishing gradient problem, as while vanilla RNN's have the backpropagation happen as an accumulating product of gradients, the LSTM mitigates this by introducing additive dynamics through the additive pointwise operator, which together with the gates allow for a better "control" over the gradient's magnitude over time.

2.2.3 Word embeddings

Big data comes in a myriad of different shapes and sizes, and the applications of deep learning to mine valuable information from this data are endless, however, a lot of this data will have to go through some sort of processing in order to be compatible with the computations of a neural network. This is especially true for textual data and NLP tasks, because as is visible from the information presented in sections 2.2.1 and 2.2.2, neural networks only process numerical data. This creates the need for effective numerical representations of textual information in order to utilise neural networks for tasks like sentiment analysis.

Word Vectors

One of the most used widely used techniques for representing text numerically is what is known as Word Vectors. As the name implies, it subsists of mapping words to real number vectors, with the idea being that the lengths and directions of these vectors in a substantial way are able to encompass the linguistic meaning of the word. A number of models for creating these mappings already exists, one of the more widely used being Word2vec[1] that was developed by a Google team consisting of Mikolov et al. (2015). Word2vec is actually a shallow neural network itself, and uses a variety of information retrieval techniques in order to produce vectors that work exceptionally well for capturing relationships between the words in the corpus. Querying the model one can find such algebraic relationships as $vec(king) - vec(man) + vec(woman) \approx vec(queen)$ or relationships between words expressed as displacements such as $vec(Germany) - vec(Berlin) \approx vec(France) - vec(Paris)$ (Wang, 2014).

Embedding layers

As an alternative to, or even a compliment to, using models like Word2vec for generating word embeddings one could introduce an embedding layer as an input layer for the neural network. An embedding layer takes the index of some word in a corpus, and produces a vector to be processed by the succeeding layers. The length of the vector that is produced as well as the size of the corpus is set when initially configuring the layer. When the layer is created an embedding matrix is initialised with random vectors, and will work as a lookup table mapping the incoming indexes to it's vector. The layer learns the same way other as layers, with the values of the vectors just being treated as additional parameters to be tuned during backpropagation.

When building a neural network for NLP tasks with an embedding layer one typically uses the all the text in the training data to construct a corpus, often in the shape of a hashmap or a similar data structure in order to give each word in the corpus a unique index. The embedding layer is configured using the size of the corpus, the size of the output vectors, and the size of the input vectors. The mechanics of a sample embedding layer can be seen in figures 2.9, where the layer is fed a vector consisting of the indexes corresponding to the words in the sentence "the boy runs fast". The indexes are used to look up their corresponding word vectors that are sent as input to the next layer in

[1] https://code.google.com/archive/p/word2vec/

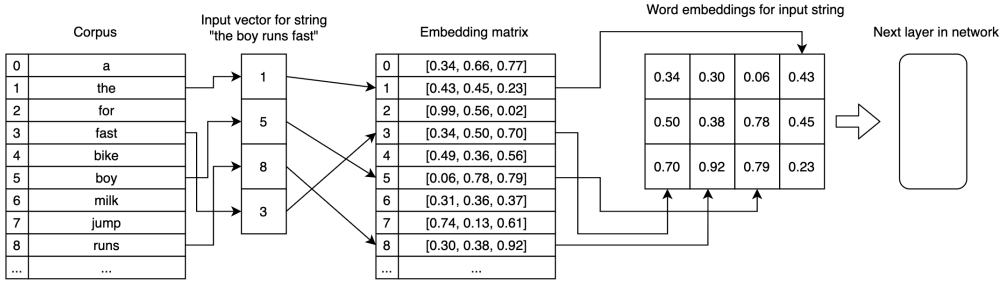


Figure 2.9: An illustration of the mechanics of a sample embedding layer.

the network, e.g. an LSTM layer that will process the inputs in a recurrent fashion as explained in section 2.2.2. We notice the increased dimensionality as the embedding layer receives a 2D vector of indexes and produces a 3D sequence of word vectors that are fit to be processed by a suceeding recurrent layer.

2.3 GPU-Paralellization

The following sections on GPU-Paralellization are also heavily based on this thesis' preliminary project (Moss, 2019), as a review of the literature was done with the conclusion that not much else has to be included to support the work done in this thesis.

Previously in traditional computing CPU's were often known as "the brains" of a computer, being responsible for the majority of the calculations and instructions in computer programs, while the GPU's tasks were limited to rendering images on a display. However, for the last two decades there has been a paradigm shift inside the field of computing, from enhancing processor clockspeed to enhancing parallelism. In other words, the way to make programs run faster is to increase it's parallelization (Macedonia, 2003). While the majority of modern CPU's today are multi-core processors, most containing between 2 and 32 cores, the numbers dull in comparison to a GPU which may contain a number of cores in the thousands. It is definitely possible to do parallel computing on a CPU, however, one may find oneself to be quickly running out of threads if the computational task is sufficiently demanding, and in such cases the programmer should investigate the possibilities of solving the problem using GPU-programming.

The GPU's many cores gives it a massive potential for multi-threading and high throughput, thus making it ideal to handle large amounts of data and well suited for tasks inside the fields of Information Retrieval and Data Mining. However, as can be deferred from figure 2.10 and figure 2.11, there are a lot of architectural and design differences between the two processors, therefore, a switch from problem solving on the CPU to the GPU requires a switch in programming paradigms and new algorithmic implementation approaches. One could say that what the GPU gains in parallelizability, it pays for with a loss of "intelligence", as it is limited to computing using primitive data types and data structures. In other words, one cannot expect to take the same program running on a CPU, run it on a

CPU	GPU
Tens of cores	Thousands of cores
Low latency	High throughput
Thrives at serial processing	Thrives at parallel processing
Large caches	Smaller caches
Complex control logic	Simple control logic

Figure 2.10: High level feature comparison between the CPU and GPU

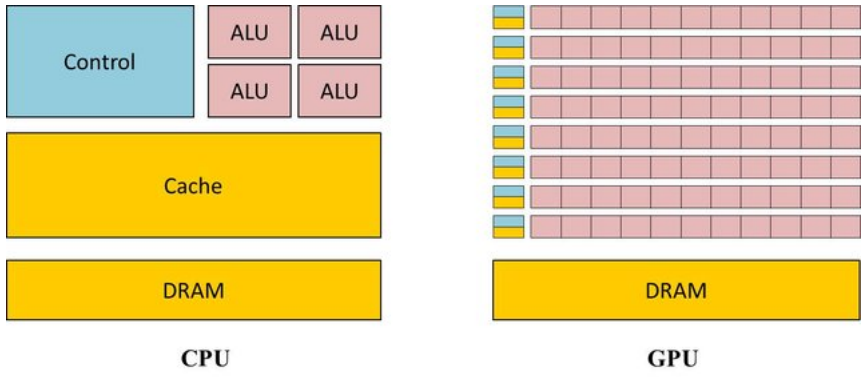


Figure 2.11: A high level architectual comparison between the CPU and the GPU. Figure retrieved from (Ben Amor, 2016).

GPU, and expect to see a performance increase ”just-like-that”. GPU-programming is an art in-and-of itself, requiring it’s own platforms and frameworks to exploit the hardware advantages present. Furthermore, we more often than not see the data we want to process in any given situation either live in CPU memory, or being read from disk by the CPU, causing extra overhead moving data from the CPU to the GPU and back for heterogeneous systems.

2.3.1 CUDA

Today the two main platforms for GPU-programming are the OpenCL[2]-platform (Open Computing Language) built by Khronos as a standard for parallel programming of heterogeneous systems, and CUDA[3] built by NVIDA as a parallel computing platform and programming model which aspires to make using a GPU for general purpose computing simple and elegant. While CUDA is widely used for Artificial Intelligence, it is also limited to running on NVIDA hardware, while OpenCL is an open industry standard and has less restrictions both when it comes to running on GPU’s from different vendors and when it comes to the type of hardware combinations as it can run on heterogeneous systems.

The vast number of cores in a GPU makes it especially efficient at basic linear alge-

[2]https://www.khronos.org/opencl/
[3]https://developer.nvidia.com/cuda-zone

bra operations as the computations involved often are linearly independent and therefore carry a huge potential for parallelization. This is an area of computing where CUDA has made significant advancements. Most notably Bell and Garland (2008) developed several CUDA-based techniques for sparse matrix-vector multiplication by extensively exploiting parallelism to utilize an impressively high fraction of the computational resources available. They found that their CUDA-based approach outperformed the state-of-the-art CPU systems for both single and double precision computations. The suitability for fast linear algebra computations is what have propelled CUDA to the forefront of deep learning and AI, and has made the company behind, Nvidia, pioneers in researching the two (Jiang, 2019). Today, most frameworks, including Deeplearning4j which is used in this thesis, contain the option to use a CUDA-backend in order to GPU-accelerate the underlying algebraic computations. These backends usually exist in the form of bindings programmed in the language of the framework.

Chapter 3

Survey

This chapter will survey work related to the most important components of this thesis. It will begin broadly by looking at other work and tools related to big data management and processing and mining of big data, with a special focus on sentiment analysis in order to survey state-of-the-art systems for this task. The scope then will narrow down to look at similar work done with the specific BDMS AsterixDB. The scope will then narrow down further to deep learning as a processing technique. First looking at work done on deep learning for sentiment analysis in order to survey performance in terms of accuracy, then moving on to survey performance in terms of speed by looking at work research of GPU-powered deep learning and deep learning frameworks.

3.1 Related work on big data

3.1.1 Distributed big data systems

The need for systems for storing and processing of big data has been around for long enough for a significant amount of research, generalized tools and specialized task-specific architectures to emerge. Skuza and Romanowski (2015) sidesteped the use of any framework and built their own distributed big data environment for sentiment analysis of twitter data in order to predict stock prices. Their system took a probabilistic approach, using Naive Bayes together with the Bag-of-Words information retrieval model and the Map Reduce programming model in order to perform their analysis. Their experiments did not include any analysis of processing speed, however, the system predicted prices remarkably similar to the real stockprices in real time. A similar approach was taken by Khuc et al. (2012) who utilized Map Reduce using the widely utilized Hadoop[1] big data platform for processing in combination with the non-relational distributed database HBase[2] in order to build a scalable distributed system for sentiment analysis of tweets. The system featured

[1] https://hadoop.apache.org/
[2] https://hbase.apache.org/

a lexicon builder and a sentiment classifier as their main components and achieved an accuracy of 73.1% classifying just under 400'000 tweets in about 5 minutes on a cluster of 5 machines. Twitter itself also utilized Hadoop for their first implementation of an engine that was meant to provide users with real-time suggestions of trending queries. However, the team consisting of Mishne et al. (2013) quickly realized the architecture of Hadoop had critical bottlenecks related to the import of data and speed of the Map Reduce jobs preventing it from achieving the speed requirement for such an engine. The team ended up instead having to create a custom in-memory processing engine specifically designed for the task in order to handle the high velocity nature of the big data being processed. For the team at twitter, the at-the-time State-of-the-art Hadoop was insufficient and suggesting the need to look beyond the Map Reduce paradigm and develop tools better suited to handle highly volatile and high velocity data in addition to high volume.

2014 saw the birth of Apache Spark[3] as a fast unified analytics engine for processing of big data, and since then it has been augmented to include both a library for stream processing and for machine learning, making it both a versatile and widely utilized system. Elzayady et al. (2018) investigated the use of Apache Spark to perform sentiment analysis of tweets, using it's machine learning library to accomplish the task with Naive Bayes, Decision Trees and Logistic Regression. Their experiments showed F-measures of 0.78 for both the Naive Bayes and Logistic Regression classifiers on a preprocessed dataset of 200'000 tweets, and a scalability analysis revealed faster execution with an increasing amount of nodes, just short of 300 seconds for training and inference on a 3-node setup. Although a powerful processing engine, Spark needs a connection to a database in order to provide persistence, leading to a more "glued-together" system with more development work to get it up and running. The distributed NoSQL database Cassandra[4] is often a go-to choice for accomplishing this as it scales gracefully with read and write throughputs being considered state-of-the-art, while also providing easy tuning between availability and consistency through it's node consensus mechanism.

3.1.2 Datamining with AsterixDB

However versatile Spark is, it's still a notch below AsterixDB[5] which in addition to being able to perform processing of both stored and streamed data also provides persistence out of the box. Alkowaileet et al. (2018) showed how AsterixDB integrates seamlessly with a multitude of machine learning libraries in order to enable data analysts with easy management of end-to-end analytical dataflows, with scale-out and speed-up experiments showing how the system scales elegantly when increasing workload and amount of nodes. In their paper describing the AsterixDB framework for stream processing and continual data ingestion, Grover and Carey (2015) performed a comparison with the popular combo MongoDB[6] + Apache Storm[7] with MongoDB being used for persistence and Storm being used as the stream processing engine. The comparison showed results favoring AsterixDB for both performance and user-experience. In his thesis, Abrahamsen (2017) per-

[3] https://spark.apache.org/
[4] https://cassandra.apache.org/
[5] http://asterixdb.apache.org/
[6] https://www.mongodb.com/
[7] https://storm.apache.org/

forms a stream processing performance and scalability analysis of AsterixDB running a machine learning based UDF in comparison with the previously mentioned Cassandra + Spark combo for the task of sentiment analysis. The experiments showed a throughput of up to 10'000 tweets processed per second for AsterixDB, however, it failed to outperform the "glued-together" Cassandra + Spark system. These results contradicted the findings of Pääkkönen (2016) who although achieved the same throughput for AsterixDB as Abrahamsen (2017), saw it outperform Cassandra + Spark. The discrepancy was likely caused by an inferior Cassandra + Spark setup and configuration done by Pääkkönen (2016), but is nevertheless a testimony to the increased complexity of "glued-together" systems. A throughput of up to 10'000 tweets was also accomplished during the thesis work of Finckenhagen (2018) who used AsterixDB as an environment to do a stream processing based comparative analysis of different CPU-powered deep learning models for sentiment analysis. The models were run with the TensorFlow java API and saw the highest accuracy of 84.02% performed by a CNN-RNN hybrid, and the highest throughput with a FCNN with a hidden layer of 64 neurons. Finally, the pre-liminary project that this thesis is based on (Moss, 2019) featured a comparison between the stream processing throughputs of UDF's using CPU- and GPU-based implementations of Naive Bayes for sentiment analysis of tweets. The CPU-based approach achieved a throughput of up to 16'000 tweets classified per second while the GPU-based UDF ran into bottlenecks related to the GPU's need for primitive datastructures.

3.2 Related work on deep learning

3.2.1 Deep learning for sentiment analysis

Deep learning has been thoroughly researched, especially during the last decade as it has seen a boom both in the of interest and progress of AI. Zhang et al. (2018) comprehensively surveyed the use of deep learning techniques and different kinds of neural networks for different uses of sentiment analysis and concluded that many of the techniques studied have shown state-of-the-art results. They talk about state-of-the-art mainly in the sense of providing the most accurate prediction compared to other machine learning techniques. The survey makes a case for a shared throne between Convolutional Neural Networks (CNNs) and RNNs, specifically LSTMs as introduced in section 2.2.2, as state-of-the-art for sentence-level sentiment analysis if used along with word embeddings such as those introduced in 2.2.3. The survey also presents works completed with both CNN's and LSTM's inside the field of sarcasm-detection, which definitely could prove useful for achieving a higher accuracy for sentiment analysis of tweets, as social media in general see a significant use of sarcasm.

Narrowing the scope from general sentiment analysis down to sentiment analysis of tweets, we take a look at work performed by Cliche (2017) at Bloomberg and their attempt to create a state-of-the-art tweet sentiment classifier, also using CNN's and LSTM's by first examining both models and then building a bona fide deep network by using both. They used bidirectional LSTM's concatenating the hidden states before running them through a fully connected layer and then getting the output through a softmax function. In other words the network read the sentence both forward and backwards before making it's mind

up as to whether it was positive or negative. Together with the CNN this LSTM setup made up what the authors called their ensemble model, which achieved the highest score out of 40 teams for the SemEval-2017[8] task 4: twitter sentiment analysis competition. Another entry in the same competition was Baziotis et al. (2017) who with their purely LSTM-based approach performed well and even tied for 1st place for subtask A. Their model stacked two siamese (shared weights) bidirectional LSTM layers and added ”attention”-layers in order to account for the relative importance of each word for the sentence's sentiment, before getting the output through a max-out layer.

Other interesting research done specifically on the use of LSTM's for sentiment analysis include Chen et al. (2018a) who used hierarchical LSTM's to perform sentiment analysis on tweets by also considering rich context features like retweets, reply history and social context. Experiments showed that their network outperformed traditional LSTM's. Chen et al. (2018b) researched sentiment analysis of tweets via the use of specially trained emoji embeddings in addition to word embeddings, along with LSTM's and attention-mechanisms similar to those of Baziotis et al. (2017). Their experiments, although completed using their own datasets, showed higher accuracy, up to 90%, than other traditionally state-of-the-art models.

3.2.2 GPU-accelerated deep learning and deep learning frameworks

As a result of the deep learning boom a myriad of different frammeworks have emerged over the last couple of years. One of the most famous ones is Tensorflow[9] which has a wide ecosystem of open-source tools and libraries for both deep learning and machine learning in general, built to scale well in large heterogenous environments (Abadi et al., 2016). Tensorflow uses static dataflow graphs to represent computation, unlike PyTorch[10] which uses dynamic graphs creating a slightly differen programming paradigm which may have contributed to it's increased popularity the last couple of years as it favors a more rapid style of prototyping and development (Paszke et al., 2017). For the data scientists preferring more abstraction, Keras[11] provides a really highlevel API for creating neural network, and provides options to run on top of other frameworks, one of them being Tensorflow. All the frameworks mentioned so far however, are all highly, if not entirely, based in python. One could suspect that data scientists often favour python due to it's simple syntax and accessibility, which might explain why so many deep learning tools have popped up for this language, however, this is a suspicion which to the best of our knowledge still yet is to be supported by research. For a java developer looking to research deep learning a natural choice is deeplearning4j[12] which is an open-source distributed deep learning framework for java and scala.

All of the frameworks and tools mentioned so far all have in common that they provide easy entrance to perform computations on the GPU. They access the GPU through bindings for CUDA libraries, as introduced in section 2.3.1. CUDA provides both general purpose computation libraries as well as more specialized ones, the most relevant being

[8] http://alt.qcri.org/semeval2017/
[9] https://www.tensorflow.org/
[10] https://pytorch.org/
[11] https://keras.io/
[12] https://deeplearning4j.org/

cuDNN[13] which is a deep neural network library of primitives specially built to speed up deep learning specific computations. Chetlur et al. (2014) showed experiments in their paper presenting cuDNN that by integrating cudNN with Caffe[14], another deep learning framework, they saw an 36% performance improvement while also reducing memory consumption. Shi et al. (2016) performed experiments comparing training times for different frameworks running single-threaded and multi-threaded on CPU's and with CUDA and cuDNN on GPU's. For LSTM-layers with 32 and 64 input neurons they saw a drop in training time from 2-4 seconds per batch down to 0.2-0.4 seconds per batch depending on the framework, a 10× speedup. Li et al. (2014) also studied GPU-accelerating RNN's, but implementing their models without any framework. Their CUDA-based GPU implementation achieved a 2-11× speed-up compared to the CPU implementations.

[13]https://developer.nvidia.com/cudnn
[14]https://caffe.berkeleyvision.org/

Chapter 4

GPU-accelerated Deep Learning inference-based UDF's for AsterixDB

This chapter will present the UDF's built for AsterixDB along with the Deep Learning model they use to perform inference and how the UDF's work to provide an optimal throughput through batch processing semantics. The chapter starts by describing the design of the deep learning model along it's optimal processing semantics for performing inference on a GPU. It then describes how this model is deployed to the final system, while giving an overview of what the final system will look like. Finally, it will move on to describe the mechanics of the UDF's and how they realize these optimal processing semantics for both stored and streamed data.

4.1 An RNN for sentiment analysis of tweets

When deciding on the architectures for the Deep Learning model for sentiment analysis, an LSTM-based RNN was chosen due to it's suitability for this task as explained in section 2.2.2 and it's promising base of research both for the task and for GPU-acceleration, including cudNN-support, as presented in section 3.2.1 and 3.2.2 respectively. We aim to build a minimalistic and lightweight, yet effective network, with the motivation being two-fold:

1. A simple lightweight network will naturally be faster as it there are fewer computations involved turning an input into an output.
2. A simple lightweight network will require less GPU resources and therefore free up more of the remaining memory to process more records in parallel.

The RNN will be realized using the deeplearning4j framework, as it will have to be compatible with the AsterixDB java UDF framework.

4.1.1 Custom embedding layer for word embeddings

Although, as discussed in section 2.2.3, Word2vec is a highly useful word embedding for capturing linguistic contexts of words, the amount of information for each word might prove excessively effective for a task like sentiment analysis where in the end a sentence will fall into one of two categories, "positive" or "negative". As the Word2vec vectors are often verbose and expensive to generate, the belief is that we can sacrifice a minimal amount of accuracy and a achieve highly efficient word embedding using sparse "home-made" word vectors by training a sparse embedding layer on the training tweets.

Pre-processing

When performing sentiment analysis, regardless of the specific algorithm, it is normal to include a pre-processing step in order to prepare the data for processing and facilitate the extraction of linguistic context. While heavy pre-processing if often utilized to achieve a higher accuracy when processing stored data, it can be a performance killer in a streaming context. It is of course impossible to pre-process a record before it is streamed into the system, and as the record arrives into the stream processing engine, if the pre-processing becomes computationally expensive enough it will significantly limit throughput. This is what choked the performance of the GPU-based Naive Bayes UDF of Moss (2019) during this thesis' preliminary project.

With this in mind the decision was made to take a minimalistic approach to the pre-processing step. The steps taken are

- Converting all words to lower-case, with the idea being that a words linguistic context is independent from it's casing.
- Switching all url's and twitter-handles (characterized by "@username") with the tokens URL and USERNAME.
- Removal of all punctuation and other non-letter characters.

While the use of n-grams and other sligthly more advanced techniques often can help yield a higher accuracy by helping encapsulate positional information of the words, we considered them too computationally expensive and "not enough bang for the buck", in addition to the fact that the RNN often captures a lot of the same information by design.

Building a word-encoding model

As explained in section 2.2.3 and evident looking at figure 2.9, in order to create an input "sentence vector" to feed into the embedding layer, we need a model that encodes words to unique integers that will be used to look up our word vectors in the embedding matrix. One could notice the inefficiency of first converting a word to an integer and then that integer to a word vector instead of just looking up the vector by directly feeding the word to the embedding layer, however, there are to the best of our knowledge no frameworks allowing this.

When building this word-encoding model we first take a pass pre-processing our training data according to the steps described in the section above, before using a tokenizer

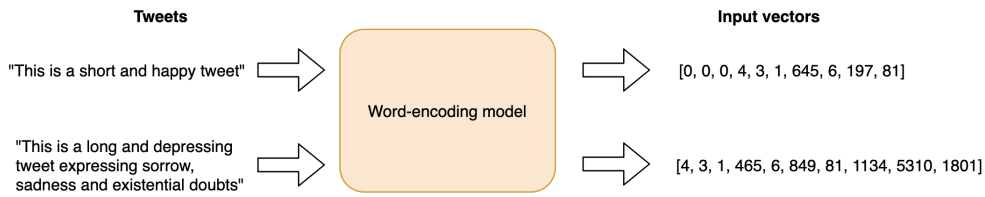


Figure 4.1: A word encoding model converting tweets to input vectors of length 10 for the embedding layer. The above tweet has less than 10 words and is padded out with 0's while the below tweet has more than 10 words and is cut after the 10th word "sadness".

to create an overview of all the tokens in all the tweets in the training data while keeping track of the their occurrence frequencies. We then we create a HashMap mapping each token t to an unique integer beginning with the most frequent token t_1 at integer 1, then increasing the integer ascendingly as the frequency of the token descends. Example of such a HashMap can be seen in listing 4.1, where we notice that highly used tokens such as *a* and *to* have low indices while less used words such as *peristeronic* have high indices.

```
{
  "a": 1,
  "to": 2,
  ...
  "acnestis": 145839,
  "peristeronic": 145840
}
```

Listing 4.1: Illustrating example of a word-encoding model generated from some training data

Embedding layer design

When designing an embedding layer we need to consider the length of the input vector, the length of the output vector(s) and the size of our embedding matrix. The **input vector** will be integers of the words that make up the tweet, generated using a word-encoding model such as described above. Because not all tweets contain the exact same number of words we have to decide on a vector length v_{input} and cut off tweets with more words than v_{input} and pad tweets with fewer words than v_{input}. When cutting off a tweet we simply add the integers corresponding to the first v_{input} words, while when padding we convert the words in the tweet to integers and prepend 0's until the vector is of length v_{input}. Figure 4.1 displays conversion of tweets to input vectors while showcasing the concepts of cutting and padding. While in terms of accuracy, padding seems like a more favorable strategy than cutting as it preserves all the information in the tweet, keeping the input vector short can make for a lighter and faster network. As such, a decision on the length of the input vector will wisely be taken with regards to the distribution of the tweet lengths in the training data.

The size of **the embedding matrix** will simply be 1 higher than the size of the word-encoding model, as it will contain all the possible words in addition to the 0 used for padding.

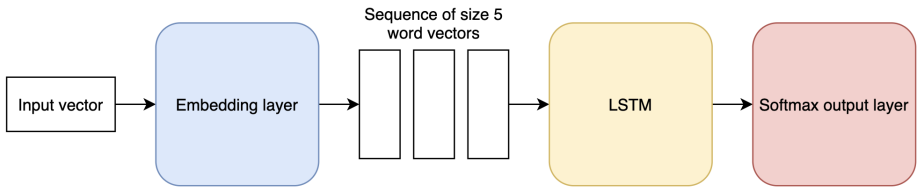


Figure 4.2: The first contribution of this paper, a lightweight highly efficient RNN for sentiment analysis.

As mentioned in the beginning of this section, we want the embedding layer to output short minimalistic word vectors that are trained specifically to help determine the sentiment of a tweet while being light enough to allow for a fast network with a high throughput. Therefore a vector length of 5 was chosen for the **output vectors** of the embedding layer. This is drastically shorter than e.g. the pre-trained Google News Word2vec model[1] which features a vector length of 300. The hypothesis is that this will constitute a significant speedup compared to networks using the Google News Word2vec model and other more complex models, while still being able capture enough lexical-sentimental information to maintain a high accuracy.

4.1.2 RNN design

The Embedding layer described in the section above constitutes the input layer for the RNN and the rest of the network follows the same design principles aiming for a minimalistic, light, fast and highly parallelizable network. As explained in the section above, the word-encoding model will encode a tweet to an input vector compatible with the embedding layer, which in turn will process the input vector and turn it into a sequence of compact word vectors of length 5. It then follows naturally that the succeeding LSTM-unit also takes in a vector of length 5. The LSTM also outputs a vector of length 5 into the output layer which utilizes the Softmax activation function as presented by equation 2.5 in section 2.2.1. Figure 4.2 shows a high-level overview of the entire network. The model uses the Multi Class Cross-Entropy loss function as described by equation 2.6 and the Adam optimizer as described in section 2.2.1.

4.1.3 Neural network parallel inference

As demonstrated in section 2.3, the GPU has a massive potential for parallelization and is especially fast at homogenoeus linear algebraic computations, which all should be exploited in order to achieve a speedup. Revisiting equation 2.1 in section 2.2.1, we look at the activation of a single neuron as a function to be determined by running the value of

$$z = \boldsymbol{w}^T \cdot \boldsymbol{a} + b$$

[1] https://code.google.com/archive/p/word2vec/

through an activation function, letting $\boldsymbol{a}$ denote the activations of the previous layer as opposed to $\boldsymbol{x}$ which we used in section 2.2.1. As most of these computations will be exceedingly consubstantial, we try to paralellize them and perform all such neuron output calculations of a given layer by

$$\boldsymbol{z}^{[l]} = \boldsymbol{W}^{[l]} \cdot \boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]} \tag{4.1}$$

and

$$\boldsymbol{a}^{[l]} = a^{[l]}(\boldsymbol{z}^{[l]}) \tag{4.2}$$

letting a superscripted $[l]$ signalize affiliation with layer l, with vector $\boldsymbol{a}^{[l]}$ being all the outputs of the neurons in a given layer l, function $a^{[l]}$ being the activation function at layer l determining the output values, vector $\boldsymbol{b}^{[l]}$ being all the biases of the nodes in a given layer l, and the weight matrix $\boldsymbol{W}^{[l]}$ being all the weights associated with each of the edges connecting the nodes in layer $l-1$ with the nodes in layer l. We note that even though the other variables increase in dimensionality when calculating multiple neuron outputs as opposed to a single one, vector $\boldsymbol{a}^{[l-1]}$ remains unchanged as the activation values from the previous layer are the same regardless of which set of edges they are multiplied with.

Aiming to parallelize further, we want to extend our computations to look at the outputs for a given layer for multiple inputs at the same time, enabling us to perform parallel inference and thereby classify multiple tweets simultaneously. Still considering a layer l we re-write equations 4.1 and 4.2 respectively as

$$\boldsymbol{Z}^{[l]} = \boldsymbol{W}^{[l]} \cdot \boldsymbol{A}^{[l-1]} + \boldsymbol{b}^{[l]} \tag{4.3}$$

and

$$\boldsymbol{A}^{[l]} = a^{[l]}(\boldsymbol{Z}^{[l]}) \tag{4.4}$$

with the matrices $\boldsymbol{A}^{[l]}$, $\boldsymbol{A}^{[l-1]}$ and $\boldsymbol{Z}^{[l]}$ constituting batches of the record-specific vectors $\boldsymbol{a}^{[l]}$, $\boldsymbol{a}^{[l-1]}$ and $\boldsymbol{z}^{[l]}$ batched together, while we notice that the network specific values of $\boldsymbol{W}^{[l]}$ and $\boldsymbol{b}^{[l]}$ remain the same regardless of what record is fed into the network. We recognize that performing parallelized deep learning inference in the end comes down to a series of matrix operations which, as established in section 2.3, the GPU is highly optimized for.

Further motivation for performing parallel inference is provided by the fact that there exists latency involved in the process of moving data from the CPU to the GPU, causing massive overhead in heterogeneous systems such as ours if we were to move and process tweets one at a time.

4.1.4 Deeplearning4j Implementation

In order to as seamlessly as possible realize the deep learning model described in this section with the AsterixDB java UDF framework, the model will be implemented, trained and used for parallel inference through the deeplearning4j framework. The framework allows models to be created both through a descriptive sequential layer-stacking model as seen in e.g. Keras, as well as through computation graph models as seen in e.g. PyTorch or

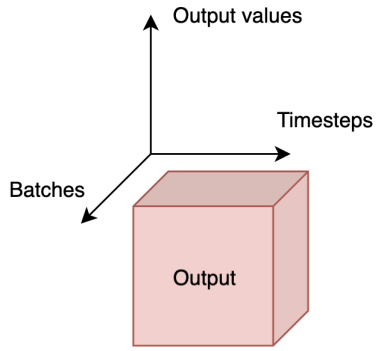


Figure 4.3: A graphical representation of the output of batches processed in an RNN in the deeplearning4j framework.

TensorFlow. After implementing the model as through the sequential paradigm as a *MultiLayerNetwork*[2] object, the model is trained on the training data before being converted to a *ParallelInference*[3] object used to perform, as the name implies, parallel inference, using the hardware to process batches of records in parallel. When running these batches through the network, the output is represented as an N-dimensional array as depicted in figure 4.3, with the different records in the batch on one dimension, the outputs at each RNN-timestep on another dimension, and the different output values, i.e. the probability of the tweet being positive and the probability of the tweet being negative, on the last dimension. This creates the need to slice out the appropriate sub-array only containing all values at the last timestep as this is the "final verdict" of the network, along with using an ArgMax-function on the output value dimension in order to get the index of the node with the highest probability value so that we can determine if the tweet was classified as positive or negative.

4.2 Model deployment and system overview

The deep learning model described in the previous section will follow a train-offline-deploy-online pattern, facilitated by the deeplearning4j framework offering model persistence through serialization and deserialization of the neural network. This method is advantageous as it saves time not having to train the network during the initalization process of the system. Furthermore, in the case that we were to discover a more effective RNN-design, it allows for easy re-deployment of the RNN without re-installing the UDF and potentially stop ongoing processing.

Figure 4.4 depicts a high level overview of our deep-learning-for-big-data system and how it's components interact with each other. The left hand side of the AsterixDB unit

[2]https://deeplearning4j.org/api/latest/org/deeplearning4j/nn/multilayer/MultiLayerNetwork.html
[3]https://deeplearning4j.org/api/latest/org/deeplearning4j/parallelism/ParallelInference.html

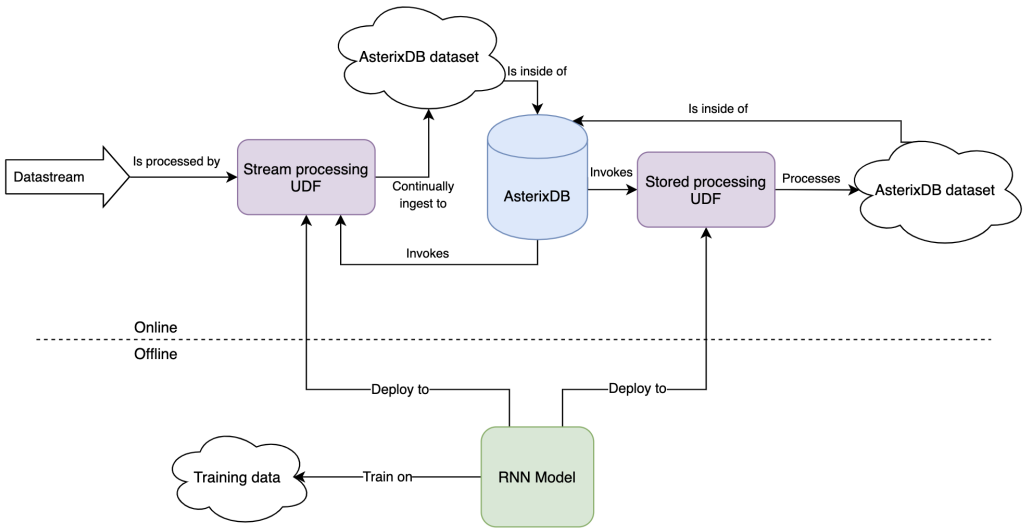


Figure 4.4: A high level overview of the system.

shows how big data stream processing is accomplished by having the UDF process a datastream while continually ingesting the processed records to an AsterixDB dataset. The right hand side of the AsterixDB unit shows how a UDF is invoked on an AsterixDB dataset as a means of processing of stored big data.

4.3 UDF design: Optimizing processing semantics

While performing parallel inference batch processing with a model such as described in section 4.1 is relatively untroublesome to accomplish in a stand-alone deeplearning4j application, it also has to be realized through the AsterixDB UDF framework in order to be suitable for the desired processing of stored and streamed big data. The AsterixDB UDF framework out-of-the-box exercises a predilection for accessing records on a ”per-record” basis, with AsterixDB handling the parallelism itself at runtime. This elicits the need for a slight paradigm shift in the way we develop UDF’s, as we must actualize a way for the framework to provide access to multiple records simultaneously in order for them to be passed as a batch to the model.

4.3.1 Batch processing stored data in AsterixDB

Normally in the case of processing stored data, a UDF is invoked from the AsterixDB query interface as shown in listing 2.4 back in section 2.1.1. Doing it this way invokes the UDF in a sequential manner, giving it access to a single record at a time on the framework

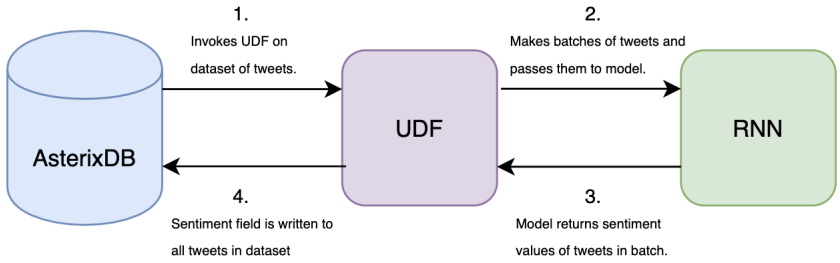


Figure 4.5: The execution flow of performing deep learning parallel inference on stored data in AsterixDB, illustrated with steps 1-4.

level while letting the corresponding Hyracks jobs be handled in parallel on the CPU if it allows for optimization that way. In order to access multiple records at the framework level we use a *group by*-statement as seen in the SQL++ listing 4.2 which causes all records in the dataset to be grouped together into a list variable that is passed to the UDF. When creating a UDF with a list-input we must use the special *[variable]* list notation for the argument and return types in the metadata function descriptor as shown in listing 4.3.

```
1 -- Enter Dataverse
2 USE TweetSentimentAnalysis;
3
4 -- Run UDF on a list of all tweets in UnprocessedTweets
5 SELECT library#RNNclassifyTweets(ARRAY_AGG(t)) FROM UnprocessedTweets t
      GROUP BY nil;
```

Listing 4.2: Calling a UDF on a list of all the records in a dataset.

```
1 <libraryFunctions>
2     <function_type> SCALAR </function_type>
3     <name> RNNclassifyTweets </name>
4     <arguments> [Tweet] </arguments>
5     <return_type> [Tweet] </return_type>
6     <definition> org.apache.asterix.external.library.
7         StoredDataLSTMSentimentFactory
8     </definition>
9 </libraryFunction>
```

Listing 4.3: Defining a UDF that processes a list of records.

Having the input be a list of tweets, we program the UDF to loop through this list and create batches realized through arrays of a suitable amount of tweets to be processed in parallel. This *batch size* can be set either as a static variable inside of the UDF code or as an additional input parameter for the UDF, and it's value should be set with respect to the GPU-environment where our deep learning model lives, i.e. the GPU memory available. As we remember from section 2.2.3, the tweet must be represented in a way understandable by the network, which means the UDF will have to run the tweets through the word-encoding model converting them to input vectors, that in turn will be stacked together to

form an input matrix, i.e. the numerical representation of a batch of tweets. Consecutively, as these batches are sent to the model and output objects such as described in figure 4.3 and section 4.1 are outputted, the UDF will slice out the appropriate output sentiment values and match them with the tweets in the corresponding input batch, constructing the output list that will be returned to the user once the last batch has been processed. This execution flow is illustrated in figure 4.5.

4.3.2 Batch processing streaming data in AsterixDB

When trying to accomplish batch processing semantics for an AsterixDB data stream we are unable to utilize SQL++ query functionality to access multiple records simultaneously, and must therefore rely on custom data types and in-memory data structures to function as batch-holders when building up batches. The UDF-design will depend on the semantics of the data stream itself, and in this section we propose two different stream processing UDF's for two different stream semantics:

- An ordinary stream of tweets such as the one obtained from the streaming component of the Twitter API[4] or from an AsterixDB Twitter-feed as described in section 2.1.1 and listing 2.2. These feeds are streaming individual tweets represented as JSON objects. We will refer to this data stream as the *Individual tweet stream*.
- An additional specialized stream streaming batches of tweets represented as JSON objects where one of the fields is a list of tweets such as those streamed in the Individual tweet stream. We will refer to this data stream as the *Batch tweet stream*.

The motivation behind including the additional data stream is to investigate the performance difference between the two UDF's of the two streams.

Individual tweet stream UDF

In the AsterixDB the feed ingestion framework presented in section 2.1.2 there exists a contract between the operators and connectors of an ingestion pipeline demanding that every record inputted to an operator has a corresponding output record. This creates a challenge for a data stream that streams tweets as individual records into the pipeline and aims to process them in batches on the compute operators.

```
1 -- Enter Dataverse
2 USE TweetSentimentAnalysis;
3
4 -- Create a datatype TweetBatch
5 CREATE TYPE TweetBatch AS OPEN {
6     id: int64,
7     tweets: [Tweet],
8     ?isDummy: boolean
9 };
```

Listing 4.4: Creating the TweetBatch datatype in AsterixDB.

[4] https://developer.twitter.com/en/docs

Algorithm 1 Individual tweet stream UDF

```
1:  procedure EVALUATE
2:      Tweet ← getNextTweetInStream()
3:      TweetBatchHolder.append(Tweet)
4:      if tweetBatchHolder.length >= batchSize then
5:          inputMatrix ← wordEncodingModel.process(tweetBatchHolder)
6:          RNNOutput ← RNN.process(inputMatrix)
7:          sentiments ← RNNOutput.slice()
8:          for i in 0...tweetBatchHolder.length do
9:              tweetBatchHolder.getTweet(i).setField("sentiment", sentiments[i])
10:         tweetBatch = functionHelper.getResultObject()
11:         tweetBatch.setField("tweets", tweetBatchHolder)
12:         tweetBatch.setField("isDummy", false)
13:         ingest(tweetBatch)
14:         tweetBatchHolder ← []
15:     else
16:         tweetBatch = functionHelper.getResultObject()
17:         tweetBatch.setField("isDummy", true)
18:         ingest(tweetBatch)
```

To deal with this challenge we introduce a new AsterixDB data type, the TweetBatch, as seen in listing 4.4 in order to create an appropriate output type for the UDF. We design the UDF to use an in-memory array shared between the compute operators to hold the accumulating incoming tweets until the batch is full, and then delegate the processing work to the operator that fills up the batch. Because every component of the feed ingestion pipeline requires a single scalar input and output, every UDF-call that does not fill up the batch necessarily has to return a dummy batch to be ingested to the target dataset. We define a dummy batch to be an object of type TweetBatch with an empty list of tweets and an *isDummy* boolean field set to true as an indicator. The UDF-call that fills up the batch vectorizes it's tweets with the word-encoding model and sends the corresponding input matrix to the RNN for processing on the GPU. After receiving the RNN output, every Tweet in the batch is enriched with a sentiment field before being put inside the tweets list of a non-dummy TweetBatch object. Finally the TweetBatch object will be ingested to the dataset and the shared in-memory array will be reset to start construction of the next batch. This whole idea is expressed in pseudocode through algorithm 1. In order to combat the clutter of having dummy data in our dataset we program the AsterixDB cluster to run periodic Cron Jobs sending a simple `DELETE FROM TweetBatchesDataset WHERE isDummy = true;` SQL++ query to one of the NC's. The execution flow of this individual tweet stream UDF can be seen in figure 4.6. An example of how the input and output records of this UDF might look can be seen in listing 4.5, where we imagine the incoming tweet being the last one in a tweet batch of size 50'000.

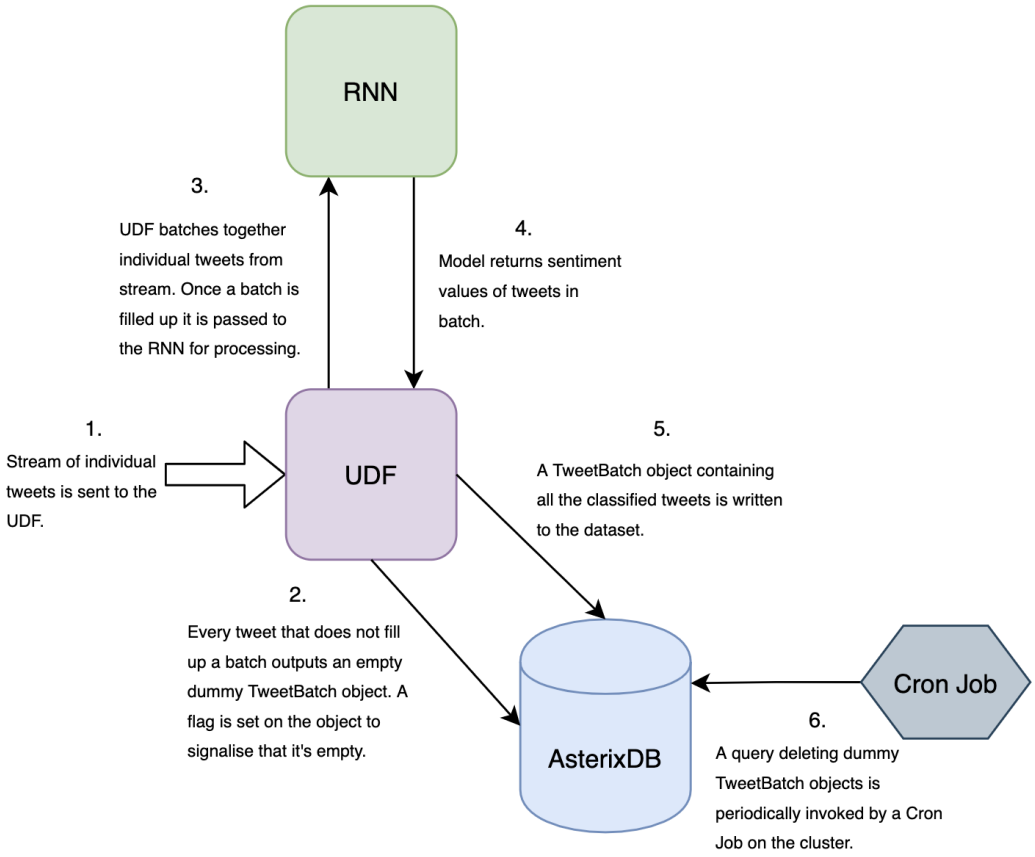


Figure 4.6: The execution flow of performing deep learning parallel inference on an individual tweet stream in AsterixDB, illustrated with steps 1-6.

```
// input
tweet: {
  id: 49999,
  text: "I love deep learning"
}

// output
tweetBatch: {
  id: 1349753,
  tweets: [
    {id: 0, text: "I love coffee", sentiment: "positive"},
    {id: 1, text: "I hate breakfast", sentiment: "negative"},
    ...,
    {id: 49999, text: "I love deep learning", sentiment: "positive"}
    ],
  isDummy: false
}
```

Listing 4.5: Example input and output values for the Individual tweet stream UDF.

Batch tweet stream UDF

When trying to realize deep learning parallel inference on a customised stream of batches the semantics become a lot more pleasant to deal with as there is no need for the UDF to do any granularity management between input and output. Using a TweetBatch datatype as the one defined in listing 4.4 as a basis for both the incoming record and the output, the UDF will always have a fixed length batch of tweets at every processing step, making for a smoother flow of data. It naturally also removes the need for dummy batches and the isDummy-field all together.

Upon receiving a batch of tweets from the stream, the UDF simply creates an input matrix from the tweets in the batch, processes it through the network, and then ingests the same batch of tweets enriched with sentiment fields for all the tweets in the tweet list. This execution flow can be seen in figure 4.7. We note a considerable reduced complexity compared to the individual tweet stream UDF of figure 4.6 due to the consistent semantics between input and output.

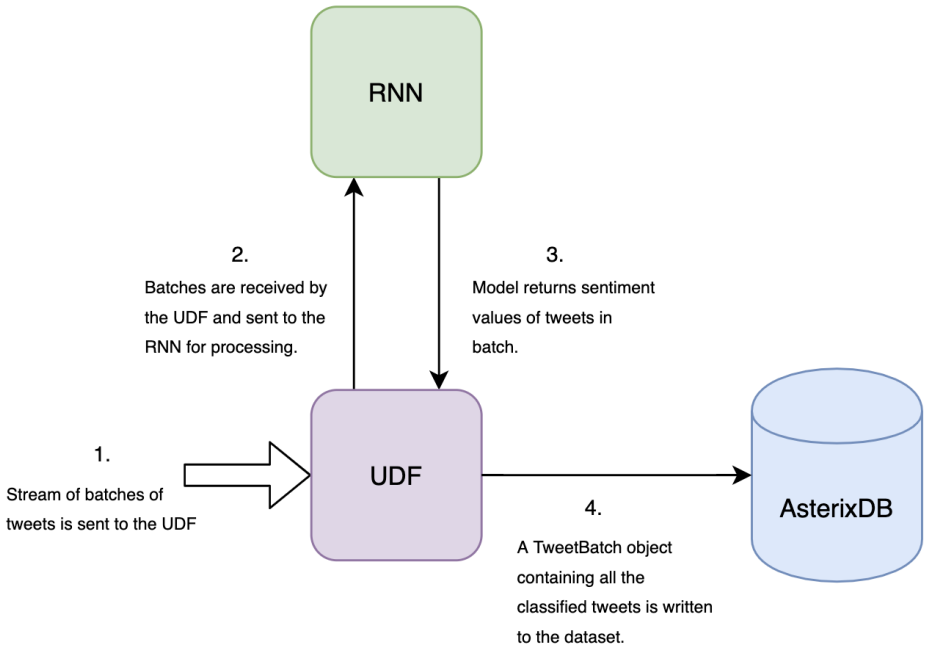


Figure 4.7: The execution flow of performing deep learning parallel inference on an batch tweet stream in AsterixDB, illustrated with steps 1-4.

Chapter 5

Evaluation of the system

This chapter will describe the setup, execution and results of the experiments conducted to investigate the performance of the UDF's developed in this thesis. It will start by going over the goals of the experiments and the environment that they will be carried out in. It will then move on to present relevant evaluation methodology like datasets used, evaluation metrics meassured, model specifics and details on how each experiment was carried out. Finally the results of the experiments will be presented.

5.1 Experimental goals

As stated in section 1.2, the goal of this research is to investigate the performance and scalability of GPU-accelerated deep learning inference bundled as a User Defined Function (UDF) for AsterixDB, as a data mining technique for stored and streamed big data. To determine this we conducted experiments gradually increasing the *volume* and *velocity* of the big data processed using the UDF's developed in section 4.3. As similar experiments have been conducted previously in AsterixDB (Pääkkönen, 2016)(Abrahamsen, 2017)(Finckenhagen, 2018)(Moss, 2019), the results will have a broad body of research for comparison of results in order to determine the viability of GPU-accelerated deep learning inference compared to other techniques for processing big data. In order to make sure the methods developed in section 4 maintains an acceptable level of accuracy we also conduct experiments measuring this.

5.2 The evaluation environment

The experiments were carried out on a machine featuring 512 GB RAM, an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz CPU and two Tesla P100-PCIE-16GB GPU's.

5.3 Evaluation methodology

5.3.1 Dataset

The Stanford sentiment140[1] tweet sentiment dataset is used to train and test the systems for evaluation. For the tweets in the dataset only the fields for sentiment and text are used as both systems use tweet text as the only input to infer sentiment, as opposed to also using information about e.g. date posted or twitter user. The dataset was created by Go et al. (2009) using the Twitter Search API for keyword searches and automatically assuming sentiment based on emoticon data, and it contains 1'600'000 tweets.

Training data

The dataset was shuffled randomly and the first 85% (1'360'000 tweets) were used to train the model for evaluation.

Test data

The remaining 15% (240'000 tweets) of the dataset were used as test data for evaluation. In order to provide *bigger* test data to better evaluate how the system perform in a big data context, the test data was duplicated 10 times to provide a total of 2'400'000 tweets for testing.

Data stream generator

In order to test the systems in a streaming context, a simple data stream generator was built using python in order to simulate a stream of tweets. The stream generator connects to the processing engines through the python socket interface[2] and can be configured to send a constant number of tweets per second, providing a consistent and deterministic environment for stream processing testing.

5.3.2 Evaluation Metrics

For machine learning tasks we normally see a trade-off between accuracy and processing speed, as a higher accuracy naturally requires more complex processing and models which require more processing power. Therefore, although model accuracy neither is the focus nor inherently inside the scope of the thesis, we still find it necessary to measure to ensure the models are not cutting corners.

The main motivation behind the experiments is to measure processing speed and throughput for the systems, therefore Tweets Processed Per Second (TPPS) will be the central evaluation metric.

[1] http://help.sentiment140.com/for-students

[2] https://docs.python.org/3/library/socket.html

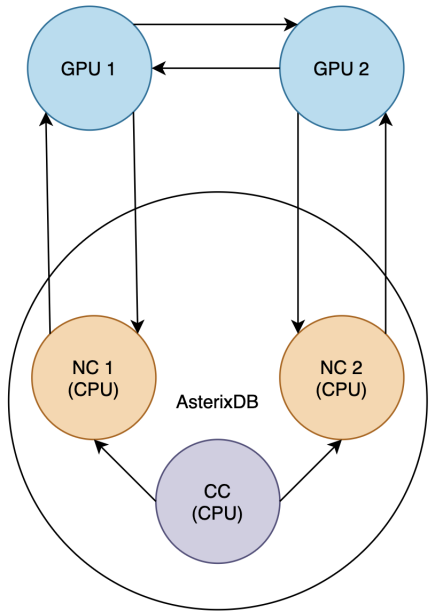


Figure 5.1: An overview of the hardware setup for the evaluation of the AsterixDB GPU-accelerated deep learning UDF's.

5.3.3 Model specifics

Configuration

As discussed in section 4.1 we have to choose an appropriate size for the input vector based on the distribution of the lengths of the tweets in the training data. A quick inspection of the data shows that $\approx$ 99.6% of the tweets contain 30 words or less. With this in mind we chose an input vector size of 30, keeping the length short enough to allow for effective processing of the majority of the tweets, while causing only $\approx$ 0.4 % of the tweets to be cut and processed without the entirety of their textual information.

Because of the vastness of the dataset we let our RNN train for 10 epochs to allow for proper fitting.

Although the GPU's memory is big enough to handle more tweets simultaneously we set the batch size to be 50'000 tweets for the Individual tweet stream UDF in order to not create congestion in the communication between the CPU and the GPU and risk running out of memory on any of the NC worker nodes. The Batch tweet stream UDF will receive batches of a certain size and pass the same batch to the GPU.

Hardware setup

For our experiments we simulate a small AsterixDB cluster on the CPU using 3 cores to spin up one CC master node and two NC worker nodes. As the worker nodes are

responsible for UDF execution they will be tasked with creating batches, sending them to their respective GPU's for processing, then receiving and writing the results to the datasets. The deeplearning4j framework facilitates loadbalancing between the two GPU's, allowing for delegation between them should any one of them have a significantly bigger load at any given time. A figure displaying the hardware setup for the evaluated AsterixDB system can be seen in figure 5.1.

5.3.4 Experiments

In addition to the investigating the accuracy of the system's classification of the testing data through the deeplearning4j *Evaluation* class, there will be two main categories of experiments in order to evaluate the performance of the systems in two different big data contexts. The first category of experiments will focus on evaluating the performance and scalability of how the system processes stored big data. The second category of experiments aims to evaluate the performance and scalability of the stream processing capabilities of the system.

When AsterixDB compiles the queries for the experiments down to Hyracks jobs, the actual processing and UDF execution will happen at the compute operator. We will use this as a basis for generating the results of the experiments, having the UDF's write system timer values to the NC-logs in order to determine processing throughput. This is done under the assumption that the UDF processing is the most computationally demanding step of the Hyracks job and therefore the place we will want to assess throughput. As noted by Abrahamsen (2017) the UDF is also the most logical operator for reporting results as gaining insights of other parts of the job will require tinkering with the underlying source code and recompiling the whole system.

Experiments on stored data

For the stored data experiments a dataset was created inside of AsterixDB through the query interface, and then filled with the 2'400'000 records from the test data. To run the experiments we used the query of listing 4.2 to invoke the UDF on the whole dataset as a list, and measured the processing time of the UDF. The process was then repeated three times more with 4'800'000, 7'200'000 and 9'600'000 tweets in the dataset in order to see how the UDF performance scales with the increasing volume of data. To not risk running out of memory and allow the Hyracks group-by operator to aggregate the whole dataset into a list-format, we increase it's memory to 33GB through specifying `compiler.groupmemory = 33000000000` in the *common* configuration. To measure throughput the UDF was programmed to write to the NC-logs the value of the system timer as it initializes before it starts processing and the value once the processing is finished.

Experiments on streamed data

For the stream processing evaluation, experiments were conducted for both of the stream processing semantics described in section 4.3 and their respective UDF's. As the two different semantics deal with different data types, namely the *Tweet* and the *TweetBatch*

types, two streams were created to stream each type and each were pointed to their own empty TweetBatch dataset. The two streams were initalized and connected to their respective datasets with their respective UDF's applied. To evaluate how each of the UDF scaled on big data with increasing velocity we used the data generator described earlier in this section to stream the 2'400'000 tweets from the test dataset multiple times while increasing the velocity each time. The velocities streamed were 10'000, 25'000, 50'000 and lastly 75'000 tweets per second which was as fast as the data stream generator could go. To measure throughput the UDF's were programmed to write to their NC-logs the value of the system timer every time a batch was processed.

5.4 Results

This section will display the results of the experiments in the order they were presented in the previous section. It starts displaying results of the accuracy-related metrics before moving on to display results of experiments done to assess throughput first on stored data processing, then on streamed data processing.

5.4.1 Results for accuracy experiments

Accuracy-related metrics for the UDF's RNN were obtained through classifying the test data using the deeplearning4j framework in a stand-alone context. Table 5.1 shows the confusion matrix of the RNN's classification of the 240'000 tweets in the test dataset, with the leftmost column displaying the ground truth classes and the top row displaying the predicted classes. Table 5.2 shows the accuracy, precision, recall and F1-scores for the RNN's classification of the 240'000 tweets in the test dataset.

	Positive	**Negative**
Positive	100388	19612
Negative	26028	93972

Table 5.1: Confusion matrix for the RNN's classification of the 240'000 tweets in the test dataset, provided by the Deeplearning4j *Evaluation* class.

Accuracy	0.8098
Precision	0.8273
Recall	0.7831
F1-score	0.8046

Table 5.2: Scores for Accuracy, Precision, Recall and F1 for the RNN's classification of the test data, provided by the Deeplearning4j *Evaluation* class.

5.4.2 Results for experiment on stored data

Table 5.3 and figure 5.2 show the results of the experiments measuring execution time for classifying stored datasets of size 2.4, 4.8, 7.2 and 9.6 million tweets. Table 5.3 displays

a stable throughput ($SD = 699.03$) between the different dataset sizes, suggesting linear scalability. In figure 5.2 the least squares method were used to obtain a linear regression line between the data points with $Error \approx 0.0228$ indicating an accurate fit and further suggesting linear scalability on an increasing volume of data.

Tweets processed	Seconds	Average Throughput (TPPS)
2 400 000	41.09	58 408.37
4 800 000	80.41	59 694.07
7 200 000	123.76	58 177.12
9 600 000	160.89	59 668,10

Table 5.3: Results for the performance scalability experiment on stored data.

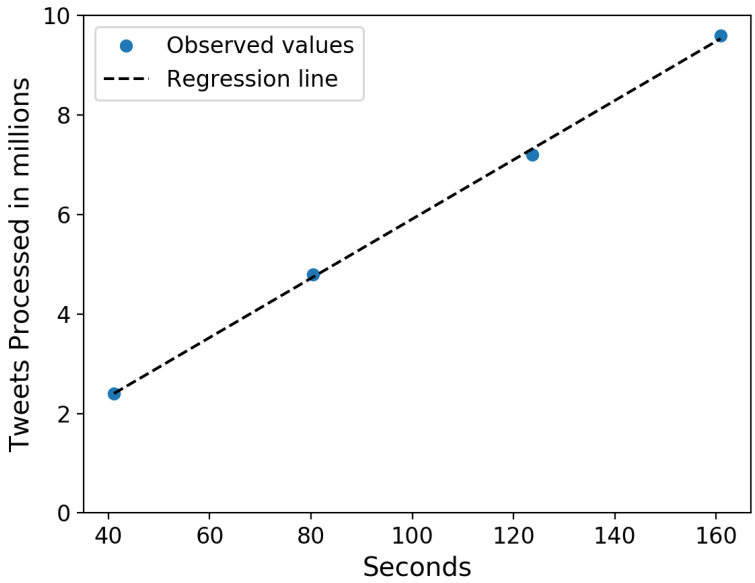


Figure 5.2: Results for the performance scalability experiment on stored data, classifying 2.4, 4.8, 7.2 and 9.6 million tweets.

5.4.3 Results for experiment on streamed data

The figures 5.3, 5.4, 5.5 and 5.6 shows the results of experiments on streamed data at 10'000, 25'000, 50'000 and 75'000 tweets per second respectively. Because the processing happens in batches it is hard to create a meaningful representation of the throughput in TPPS over time as it will depend a lot on the processing time of the individual batches, causing big fluctuations depending on whether a batch for whatever reason is processed faster or slower than normal. Therefore, we chose to graph the average throughput over

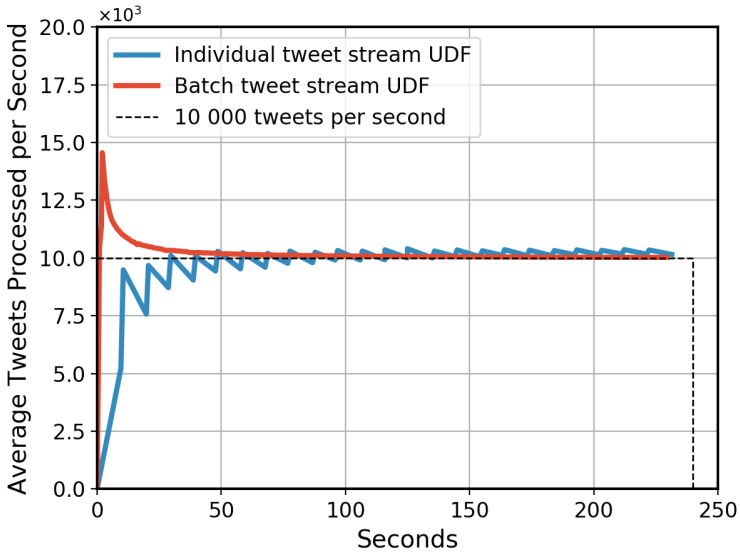


Figure 5.3: Results for experiment having the two UDF's classify a data stream of 10'000 tweets per second for a total of 2.4 million tweets.

time, meaning each point on the graphs is the total amount of records processed at that point in time divided by the total processing time. Because of this we expect to see values fluctuate in the beginning and then diverge towards a stable average value, as an average value per definition becomes harder to change as the amount of data gathered increases.

We graph the average throughputs for the two UDF's together for each experiment in order to facilitate comparisons. We also add a dotted line to represent the throughput of the data stream.

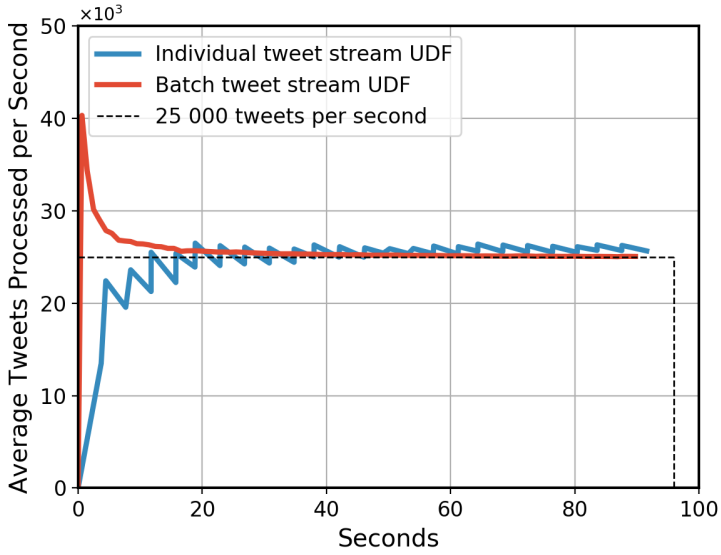


Figure 5.4: Results for experiment having the two UDF's classify a data stream of 25'000 tweets per second for a total of 2.4 million tweets.

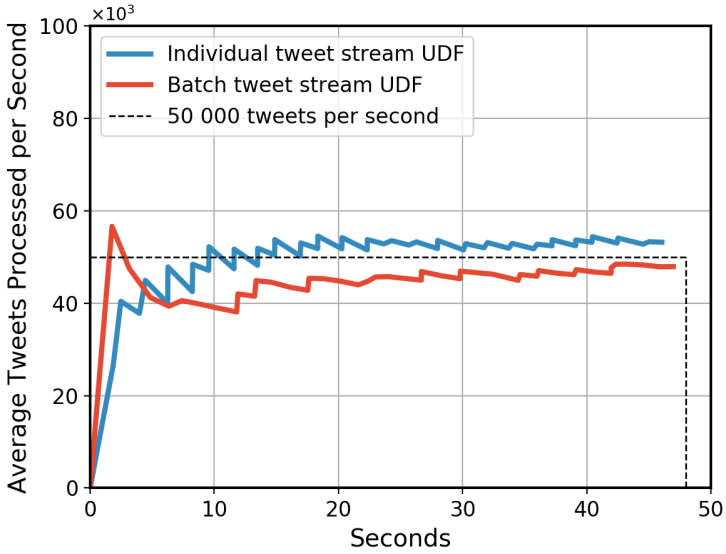


Figure 5.5: Results for experiment having the two UDF's classify a data stream of 50'000 tweets per second for a total of 2.4 million tweets.

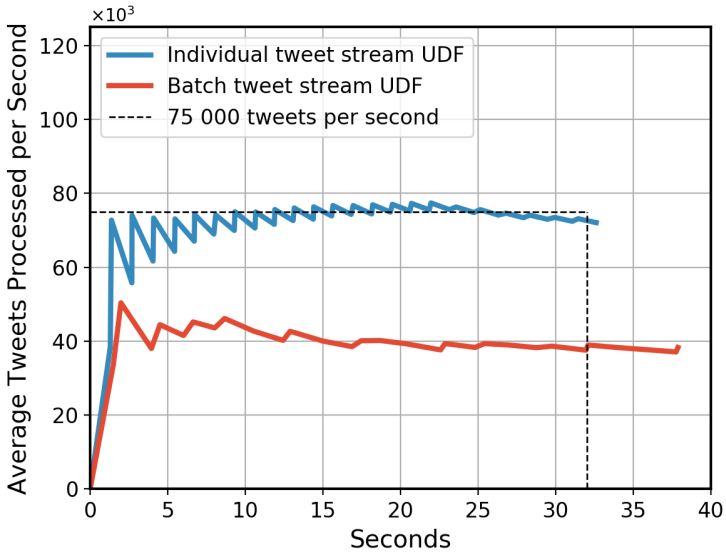


Figure 5.6: Results for experiment having the two UDF's classify a data stream of 75'000 tweets per second for a total of 2.4 million tweets.

Chapter 6

Discussion & evaluation

This chapter will discuss the results achieved in the previous chapter, as well as include a minor discussion the comparative baseline used to draw conclusions from the results.

6.1 Results

6.1.1 Model accuracy

Although achieving the highest possible accuracy was outside the scope of this thesis we still find it necessary to briefly discuss the results obtained. An accuracy of 80.98% was achieved for the model, which definitely is in the ballpark of what one should expect and consider acceptable for sentiment analysis, even though others have achieved even higher values. The confusion matrix reveals a small predilection for labeling tweets positive with a total of 126'416 records being classified as positive and 113'584 being classified as negative, in a dataset that had a half-and-half distribution. It is however, not significant enough to clearly indicate anything other than coincidence or noise.

In hindsight it is possible 10 epochs of training can have caused the model to over-fit for the training data and hence lower the accuracy for the testing data. To achieve a higher accuracy some sort of hyperparameter tuning could have been applied to find the optimal number of training epochs along with other hyperparameters such as weight initialization, activation functions, learning rate and optimization algorithm. An obvious way to boost accuracy could also be to increase the size of the embedding layer in order to capture more linguistic information or even introduce more layers in the network, however, both of these measures would likely negatively affect throughput and scalability which where the main metrics we sought out to optimize. Hyperparameter optimization or a different network structure might also have gotten rid of the tendency to classify more tweets as positive than negative.

While Abrahamsen (2017) chose to omit all accuracy related metrics when doing machine learning scalability work with AsterixDB, Finckenhagen (2018) demonstrated with their neural networks accuracies of 81.44% and 84.02% on the Stanford Sentiment140

dataset for his fastest and slowest networks respectively. The slowest network was a CNN-RNN hybrid while the fastest was a small forward connected neural network, and as the latter likely is a less complex model than the RNN developed in this thesis, a higher accuracy should have been attainable e.g. through the measures mentioned above.

6.1.2 Stored data processing

The experiments on stored data displayed in table 5.3 revealed a stable throughput of between 58'000 and 60'000 TPPS, with a standard deviation of 699.03 being indicative of a consistent throughput with some noise. The stability of the measured throughput values demonstrate a linear scalability for an increasing volume of data, which is further supported by the regression line of figure 5.2. Although not specifically measured, we clearly noticed an increasing delay from the query was sent until the actual processing started for the bigger datasets, likely due to a growing computational load for the Hyracks group-by operator as the size of the datasets increased. Should this be the case the group-by operator will likely provide an additional bottleneck, resulting in a decreasing lower total throughput as the dataset size increases, even though the throughput of the actual processing step remains constant. This ultimately causes an exponential growth in processing time as the size of data increases, and especially highlights the need for Big Data Management Systems and processing engines to facilitate batch processing in order to avoid unnecessary bottlenecks like these. Performing stored processing using a group-by operator in order to be able to perform parallel inference does resemble a *workaround*-like solution, and with GPU-accelerated deep learning providing a high throughput at the processing step it is imperative that big data systems allow easier integration with this technology in order to scale optimally.

6.1.3 Streamed data processing

For the stream processing experiments we compare the two UDF's processing streams of different semantics and see the Individual tweet stream UDF outperform the Batch tweet stream UDF as the we scale the velocity of the data. At first sight these results might seem surprising given that the UDF processing individual tweets has a much higher write-load having to output 49'999 dummy batches for every 50'000 tweets received, while the UDF receiving batches only has to process and output the same batch it receives, however, there are a couple of possible explanations. One of them is that AsterixDB's Central Feed Manager better handles load-balancing between the worker nodes when the records received from the stream are small. This could make for a smoother batching process receiving and pre-processing tweets as soon as they come in, while for the other UDF receiving them all in one big batch makes for a more computationally expensive extraction of tweets from the batch and delayed pre-processing before being able to send the input matrix to the GPU for processing. Further, for the 75'000 tweets per second experiment, the Batch tweet stream UDF is working with 50% bigger batches which in addition to potentially being too taxing on the GPU's, also further worsens the effect of the first disadvantage described.

We also notice a difference in the shape of the two graphs, especially for the experiments at the 10'000 and 25'000 tweets per second streams. While the Batch tweet stream

UDF has a smooth curve quickly stabilizing at the rate of the throughput of the stream, the Individual tweet stream UDF produces a more jagged curve likely due to the fixed batch size taking longer to fill up for these lower throughputs. Additionally, because of the Central Feed Manager's load balancing between the worker nodes, there likely has to arrive significantly more than 50'000 tweets in total before any of the compute operators at the worker nodes have a full batch, causing batches to be processed at even more unsteady rates, further magnifying the effect.

The way the data stream generator is set up it sends out the amount of tweets corresponding to the throughput of a second and then waits for the remainder of the second. This explains the early peak of the Batch tweet stream UDF as the time from it's initialization until it has processed the first batch is less than a second, thus giving the appearance of an initial throughput higher than the the velocity of the stream. It also explains why we on some graphs see the UDF curves stop just before the dotted line representing the stream.

The Individual tweet stream UDF achieved a maximum throughput of 75'000 TPPS which to the best of our knowledge is unlike anything previously achieved through AsterixDB for a similar setup. Abrahamsen (2017) who used the WEKA machine learning library for sentiment analysis in AsterixDB was not able to surpass a throughput of 10'000 TPPS during their stream processing experiments, even when increasing the number of worker nodes to 8 which is 6 more than the simulated cluster in these experiments. Finckenhagen (2018) who also experimented with deep learning based UDF's for sentiment analysis, though on CPU and without utilizing parallel inference, achieved a maximum throughput of 10'000 tweets per second, also on an 8-node cluster. During the preliminary project preceding this thesis (Moss, 2019), experiments were run on the very same machine as these experiments, testing the throughput of a Naive Bayes based UDF for sentiment analysis in the same streaming environment, and achieving a throughput of up to 16'000 TPPS. The superior throughputs achieved by Individual tweet stream UDF clearly validates GPU-accelerated deep learning inference as a high performance data mining technique for big data streams. It also goes to show the viability of mixed hardware systems with symbiotic CPU-GPU relationships, letting the GPU's to do most of the heavy lifting of the processing and allowing CPU's to handle the database specific tasks, as opposed to purely CPU-based clusters.

We do note, however, that the need for using dummy batches in order to accomplish the desired processing semantics also seems very much a *workaround*-like solution, similarly to the stored data processing UDF. This further emphasizes the need for big data mangagement systems like AsterixDB to facilitate batch processing semantics in order to easier adopt deep learning inference as a high performance tool for processing.

6.2 Comparative baselines

Using an experimental baseline was considered, but omitted. Both Abrahamsen (2017) and Pääkkönen (2016) compared AsterixDB to Spark + Cassandra using Spark for processing and Cassandra for persistence. While this option was considered, we concluded that the differences in the processing hardware between the systems would make it hard to yield a meaningful comparison of the effectiveness of the two systems. This was the case for most

potential baselines. To achieve using the same CPU-GPU hardware combination with a Spark + Cassandra system, and thus obtain a more fair and meaningful comparison of the systems by keeping the hardware constant, we would necessarily have to implement the same UDF's in Spark with different customizations in order to realize the same processing semantics as those used in AsterixDB, effectively doubling the development work done for the thesis and striding out of the scope of the research. However, these reflections do further support the conclusion that big data systems and processing engines should work to facilitate batch processing semantics both for processing of streamed and stored data in order to better allow the utilization of GPU-based deep learning inference.

For this reason we chose to rely comparatively on the broad body of previous research done with the AsterixDB UDF framework in order to determine the viability of the methods researched in this thesis, and we do believe that this yielded the most meaningful results. Neither Abrahamsen (2017) nor Pääkkönen (2016) achieved streaming throughputs beyond 15'000 TPPS for their Spark + Cassandra implementations, which is a number far surpassed by the 75'000 TPPS achieved for the individual tweet stream UDF in this thesis. We believe the big difference is illustrative of the power of GPU-accelerated deep learning parallel inference compared to normal CPU cluster based machine learning.

Chapter 7

Conclusion & future work

This chapter will conclude the research performed in this thesis as well as suggest future work to be carried out.

7.1 Conclusion

7.1.1 Summary and conclusion of research

This research sat out to explore the performance and scalability of GPU-accelerated deep learning inference as a data mining technique for stored and streamed big data. The research was motivated by the never ending search for faster and more scalable solutions to process and generate value from big data, as the amounts of data in the world is growing at unprecedented rates while our ability to process said data lags behind. To carry out the research we chose sentiment analysis of tweets as the data mining task to be realized through deep learning inference in order to explore it's performance and scalability, and used AsterixDB as the Big Data Management System providing an environment for persistence and processing of stored and streamed big data.

We developed a lightweight minimalist Recurrent Neural Network (RNN) with a sparse embedding layer designed to run on the GPU and perform sentiment analysis efficiently and effectively. To achieve optimal processing throughput on the GPU we utilized parallel inference to process multiple tweets simultaneously in batches, and demonstrated that this mathematically resolves to a number of linear algebra computations for which the GPU with it's high number of cores is highly suited to perform. We then developed User Defined Functions (UDF) using the RNN to classify tweets for AsterixDB in order to experimentally determine it's performance and scalability in a big data context. As the AsterixDB UDF framework operates at a per-record paradigm, creative solutions had to be applied to realize batch processing. For the stored data processing UDF we used a GROUP BY operator to pass the entire dataset to the UDF as a list, and then designed the UDF to create batches from that list. For the streamed data processing we investigated different semantics for the stream, both streaming individual tweets and streaming batches of tweets. For

the UDF processing the individual tweet stream we used a shared in-memory data structure to hold the the streamed records until a batch was filled up and could be sent to the GPU for processing, along with a custom data structure for holding and storing the batch once it was processed. For the UDF processing the batch tweet stream we used the same custom data structure, but allowed the batches to be passed to the GPU the same way they were received from the stream.

Experiments were performed with a 2 worker node cluster and 2 GPU's, scaling the volume and velocity of the data processed by the UDF's. The experiments showed linear scalability for the stored data UDF on increasing dataset sizes, ranging from 2.4 million tweets to 9.6 million tweet. There were, however, indications of potential bottlenecks earlier in the pipeline, possibly caused by an increasing load on the GROUP BY-operator as the dataset size increased. Streamed data experiments showed the individual tweet stream UDF outperform the batch tweet stream UDF, probably due to the lower payloads from the stream providing better load balancing between worker nodes as well as less overhead when pre-processing tweets to make them receivable by the RNN. The individual tweet stream UDF handled velocities up to 75'000 tweets per second, a tremendous throughput for stream processing and the highest we have seen for this type of experiments with AsterixDB. These experiments demonstrated a high viability for GPU-accelerated deep learning inference as a big data processing technique, which was what the research sat out to explore.

7.1.2 Research key takeaways

As part of our conclusion we draw the following key takeaways from the research conducted in this thesis:

- GPU-accelerated deep learning inference is a highly potent and powerful data mining technique that scales well to big data.
- To optimally implement GPU-accelerated deep learning inference one has to utilize batch processing and parallel inference as the hardware favors this type of computation. There is as well a latency involved with moving data from the CPU to the GPU and back causing a significant overhead if one were to process record individually. This creates a need for big data management systems and processing engines to better facilitate batch processing semantics in order to easier take advantage of this technology.

7.2 Future work

The experiments conducted in this thesis featured a simulated 3 node cluster on the CPU with one CC master node and two NC worker nodes who each had access to one GPU. In order to perform an even deeper scalability analysis, future work could include both scaling an equivalent cluster by adding more worker nodes and GPU's, as well as trying out different node combinations by keeping the number of NC's fixed and increasing the number of GPU's and vice versa in order to best establish the optimal combination of the two.

As Google is developing an even more specialized hardware for deep learning and AI, the Tensor Processing Unit (TPU), we leave it up to future research to compare it to the GPU and evaluate it's viability for helping further scale deep learning inference to big data.

Bibliography

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al., 2016. Tensorflow: A system for large-scale machine learning, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 265–283.

Abrahamsen, T.M., 2017. Scaling machine learning methods to big data systems. Master thesis, Department of Computer Science, NTNU – Norwegian University of Science and Technology .

Alkowaileet, W., Alsubaiee, S., Carey, M.J., Li, C., Ramampiaro, H., Sinthong, P., Wang, X., 2018. End-to-end machine learning with apache asterixdb, in: Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, pp. 1–10.

Alsubaiee, S., Altowim, Y., Altwaijry, H., Behm, A., Borkar, V., Bu, Y., Carey, M., Cetindil, I., Cheelangi, M., Faraaz, K., et al., 2014. Asterixdb: A scalable, open source bdms. arXiv preprint arXiv:1407.0454 .

Ashton, K., et al., 2009. That 'internet of things' thing. RFID journal 22, 97–114.

Baziotis, C., Pelekis, N., Doulkeridis, C., 2017. Datastories at semeval-2017 task 4: Deep lstm with attention for message-level and topic-based sentiment analysis, in: Proceedings of the 11th international workshop on semantic evaluation (SemEval-2017), pp. 747–754.

Bell, N., Garland, M., 2008. Efficient sparse matrix-vector multiplication on CUDA. Technical Report. Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

Ben Amor, N., 2016. Towards a dynamic deployment strategy of wheelchair command applications on heterogeneous architecture. Journal of Information Assurance and Security 11, 117–125.

Borkar, V., Bu, Y., Carman Jr, E.P., Onose, N., Westmann, T., Pirzadeh, P., Carey, M.J., Tsotras, V.J., 2015. Algebricks: A data model-agnostic compiler backend for big data languages, in: Proceedings of the Sixth ACM Symposium on Cloud Computing, pp. 422–433.

Borkar, V., Carey, M., Grover, R., Onose, N., Vernica, R., 2011. Hyracks: A flexible and extensible foundation for data-intensive computing, in: 2011 IEEE 27th International Conference on Data Engineering, IEEE. pp. 1151–1162.

Chen, Y., Yuan, J., You, Q., Luo, J., 2018a. Twitter sentiment analysis via bi-sense emoji embedding and attention-based lstm, in: Proceedings of the 26th ACM international conference on Multimedia, pp. 117–125.

Chen, Y., Yuan, J., You, Q., Luo, J., 2018b. Twitter sentiment analysis via bi-sense emoji embedding and attention-based lstm, in: Proceedings of the 26th ACM International Conference on Multimedia, Association for Computing Machinery, New York, NY, USA. p. 117–125. URL: https://doi.org/10.1145/3240508.3240533, doi:10.1145/3240508.3240533.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E., 2014. cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 .

Cliche, M., 2017. Bb_twtr at semeval-2017 task 4: Twitter sentiment analysis with cnns and lstms. arXiv preprint arXiv:1704.06125 .

Elzayady, H., Badran, K.M., Salama, G.I., 2018. Sentiment analysis on twitter data using apache spark framework, in: 2018 13th International Conference on Computer Engineering and Systems (ICCES), pp. 171–176.

Fan, W., Bifet, A., 2013. Mining big data: current status, and forecast to the future. ACM sIGKDD Explorations Newsletter 14, 1–5.

Finckenhagen, J.M.K., 2018. Neural networks for sentiment analysis in asterixdb. Master thesis, Department of Computer Science, NTNU – Norwegian University of Science and Technology .

Go, A., Bhayani, R., Huang, L., 2009. Twitter sentiment classification using distant supervision. CS224N project report, Stanford 1, 2009.

Grover, R., Carey, M.J., 2014. Scalable fault-tolerant data feeds in asterixdb. arXiv preprint arXiv:1405.1705 .

Grover, R., Carey, M.J., 2015. Data ingestion in asterixdb., in: EDBT, pp. 605–616.

Hochreiter, S., 1991. Untersuchungen zu dynamischen neuronalen netzen. Diploma, Technische Universität München 91.

Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al., 2001. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural computation 9, 1735–1780.

Jiang, Y., 2019. Research on the development of nvidia, in: 4th International Conference on Humanities Science, Management and Education Technology (HSMET 2019), Atlantis Press.

Keller, F., Wendt, S., 2003. Fmc: An approach towards architecture-centric system development, in: 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2003. Proceedings., IEEE. pp. 173–182.

Khuc, V.N., Shivade, C., Ramnath, R., Ramanathan, J., 2012. Towards building large-scale distributed systems for twitter sentiment analysis, in: Proceedings of the 27th annual ACM symposium on applied computing, pp. 459–464.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 .

Li, B., Zhou, E., Huang, B., Duan, J., Wang, Y., Xu, N., Zhang, J., Yang, H., 2014. Large scale recurrent neural network on gpu, in: 2014 International Joint Conference on Neural Networks (IJCNN), IEEE. pp. 4062–4069.

Macedonia, M., 2003. The gpu enters computing's mainstream. Computer 36, 106–108. doi:10.1109/MC.2003.1236476.

Mikolov, T., Chen, K., Corrado, G.S., Dean, J.A., 2015. Computing numeric representations of words in a high-dimensional space. US Patent 9,037,464.

Mishne, G., Dalton, J., Li, Z., Sharma, A., Lin, J., 2013. Fast data in the era of big data: Twitter's real-time related query suggestion architecture, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 1147–1158.

Moss, T.B., 2019. Gpu-accelerated multinomial naive bayes for sentiment analysis of tweets in a big data streaming context using asterixdb. Preliminary project leading up to master thesis, Department of Computer Science, NTNU – Norwegian University of Science and Technology .

Pääkkönen, P., 2016. Feasibility analysis of asterixdb and spark streaming with cassandra for stream-based processing. Journal of Big Data 3, 6.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A., 2017. Automatic differentiation in pytorch .

Perrons, R.K., Jensen, J.W., 2015. Data as an asset: What the oil and gas sector can learn from other industries about "big data". Energy Policy 81, 117–121.

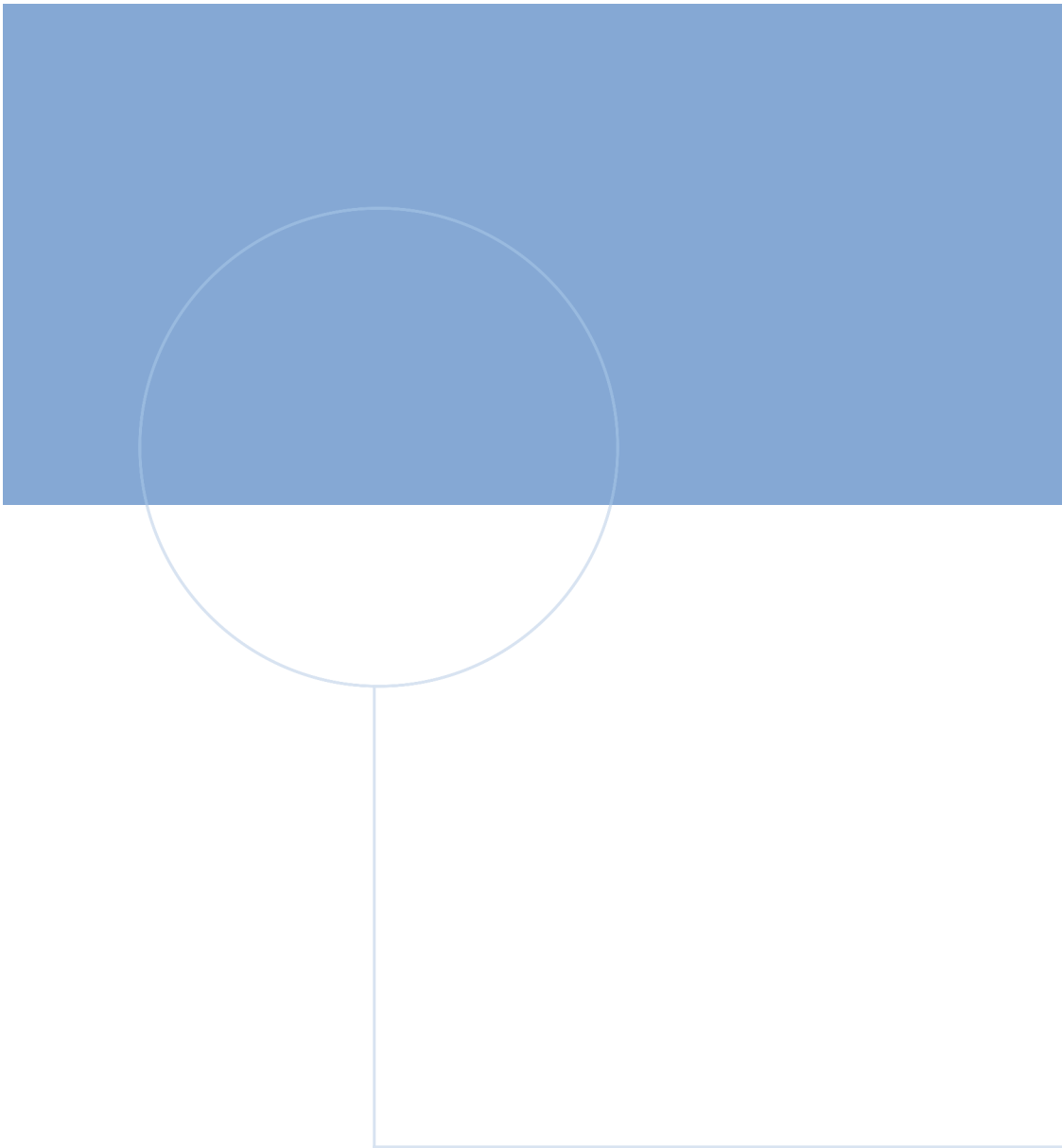
Schmidhuber, J., 2015. Deep Learning. Scholarpedia 10, 32832. doi:10.4249/scholarpedia.32832. revision #184887.

Shi, S., Wang, Q., Xu, P., Chu, X., 2016. Benchmarking state-of-the-art deep learning software tools, in: 2016 7th International Conference on Cloud Computing and Big Data (CCBD), IEEE. pp. 99–104.

Skuza, M., Romanowski, A., 2015. Sentiment analysis of twitter data within big data distributed environment for stock prediction, in: 2015 Federated Conference on Computer Science and Information Systems (FedCSIS), IEEE. pp. 1349–1354.

Wang, H., 2014. Introduction to word2vec and its application to find predominant word senses. URL: http://compling. hss. ntu. edu. sg/courses/hg7017/pdf/word2vec% 20and% 20its% 20appli cation% 20to% 20wsd. pdf .

Zhang, L., Wang, S., Liu, B., 2018. Deep learning for sentiment analysis: A survey. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 8, e1253.



NTNU

Kunnskap for en bedre verden