University of Zurich[UZH]

Predicting SPARQL Query Performance with TensorFlow

Bachelor Thesis May 10, 2017

David Ackermann
of Zürich, Switzerland

Student-ID: 11-717-311
david.ackermann@uzh.ch

Advisor: **Shen Gao**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Zusammenfassung

Während das Semantic Web zunehmend an Aufmerksamkeit gewinnt, gibt es eine Herausforderung bei der effizienten Verwaltung großer RDF-Datensätze. In dieser Arbeit behandeln wir das Problem der Vorhersage von SPARQL Anfragen mit maschinellem Lernen. Wir bauen einen Vektor, der die Abfrage strukturell beschreibt und trainieren damit verschiedene maschinelle Lernmodelle. Wir erforschen Wege, um die Leistung unserer Modelle zu optimieren und analysieren TensorFlow auf dem IFI-Cluster. Während wir bekannte Feature-Modellierungen übernehmen, können wir die Vektorgrösse reduzieren und damit Rechenzeit einsparen. Unser Ansatz kann die bestehenden Ansätze in einer effizienteren Weise deutlich übertreffen.

Abstract

As the Semantic Web receives increasing attention, there is a challenge in managing large RDF datasets efficiently. In this thesis, we address the problem of predicting SPARQL query performance using machine learning. We build a feature vector describing the query structurally and train different machine learning models with it. We explore ways to optimize our model's performance and analyze TensorFlow deployed on the IFI cluster. While we adopt known feature modeling, we can reduce the vector size and save computation time. Our approach can significantly outperform existing approaches in a more efficient way.

Table of Contents

1

Introduction

1.1 Motivation

Machine learning (ML) has been one of the most exciting research fields in the past few years. Different ML techniques have been widely adopted in various areas such as image recognition and natural language processing. The winning of AlphaGo further highlights the potential impact of ML. Among all the advantages of ML, one especially interesting point is that ML can hugely outperform rule-based or heuristic-based systems in areas like entity recognition and information extraction. Instead of letting humans specify and program complex rules, ML can automatically discover and learn the rules to achieve better performance. In SPARQL query optimization, heuristics also play a major role. Currently, most of the SPARQL engines derive query execution plans based on heuristics about the underlying data. Given the advantages of ML, this thesis will study the question of using ML to replace the heuristics in SPARQL query optimization. Specifically, this thesis will focus on predicting SPARQL query performance, such as query execution time.

1.2 Use Cases

Suppose we have an SPARQL endpoint. After a user inputs a query, the endpoint would like to provide an estimation of the query's performance, without actually executing it. We are specifically interested in estimating the execution time because it is applicable in many places:

- The user can refine the query, based on the prediction. Some users are not too familiar with SPARQL and what constitutes an expensive query. With our model, we could signal if a query is over the time-out limit and needs refinement.
- The user can have feedback on how different SPARQL commands can have different influences on the performance. As the user edits the query, we can give feedback, whether or not the new query will be executed faster. Such a feedback loop could provide a learning experience for the user.
- The provider can optimize the allocation of computing resources and therefore achieve an overall better user experience.

1.3 Problem Statement

We want to build a TensorFlow model that predicts SPARQL query performance. We have following problems at hand:

- Analyze existing approaches and feature modeling techniques in literature.
- Train a multilayer perceptron (MLP) with TensorFlow on the feature vector, then optimize and analyze the performance. See how a MLP compares to existing approaches that use clustering algorithms.
- Find different machine learning models, such as RNN that maybe solve our problem better and compare to our first model.
- Deploy TensorFlow on the IFI cluster and analyze performance.

1.4 Methodological approach

We want to approach our work in the following structure:

1. Review existing methods and techniques to predict query performance, specifically [Zhang et al., 2016] and [Hasan, 2014]. The goal is to extract which features and machine learning (ML) models are used and to find possible improvements in their approach.
2. From there we deploy a first crude MLP with TensorFlow. We then optimize this model and look for different model structures, that might bring benefits to our problem.
3. In the end, we find ways to deploy TensorFlow on the IFI cluster, which will unlock much more computing power than is available on a conventional machine. It also enables further and much more computationally expensive research.

1.5 Structure of the work

In chapter 2 we analyze two recent papers that are most relevant to our project. We go further in chapter 3 to explain our implementation. We first build a simple multilayer perceptron (MLP) and optimize it. In section 3.3 and 3.4 we explore different TensorFlow models, that are also capable of solving our problem. In chapter 4 we give an overview on some TensorFlow specific structures, which are relevant for readers that are new to TensorFlow and want to understand our code. We conclude our work in chapter 5 and 6 with a critical reflection and a summary of completed and future work.

2

Analysis of existing approaches

There aren't many papers aiming to solve the problem we have at hand, and while we have the same end goal, we take a slightly different route. It follows a short analysis of the two most recent papers that seem most relevant to our work.

2.1 Paper "Learning-based SPARQL Query Performance Prediction"

The newest paper is from Zhang et al. (2016) [Zhang et al., 2016], to which we primarily compare our work.

2.1.1 Prediction

In their model, they use a two-step prediction method (Fig.1 [Zhang et al., 2016]). First, they use Support Vector Machine (SVM) to classify the queries into four buckets: short, medium short, medium and long. As a second step, they train K-Nearest Neighbours (KNN) on each class of query. Unfortunately, they do not go into detail on how the decision to first classify the query, was derived. Also, KNN is expected to perform better on a denser dataset. Therefore we can just increase the number of buckets to improve the accuracy of each model. The choice to use exactly four buckets seems arbitrary.

Additionally, using a combination of models is expected to improve the accuracy of the result, but if that is the most efficient way to arrive at a satisfying performance is questionable. Similar to the point about the number of buckets used, why could just add another algorithm to increase the accuracy further. If we want to examine the efficiency of such combination, we would need a more detailed time analysis, than provided in 5.4, which only represents the total time for feature modeling and classification. We could therefore presumably improve their paper by just increasing the number of buckets in their classification and algorithms used.

2.1.2 Feature Modeling

They have two vectors which represent their extracted features: **Algebra Feature** and **BGP Feature**. The details of these feature vectors are further explained in chapter 3. In the paper, they have one further vector, called **Hybrid Feature**. In their hybrid vector, they use a manual feature selection with the contribution to overall prediction performance as the selection heuristic. In our approach this is not applicable: We want our model to learn which features to consider and which to ignore. It is, of course, interesting to look at the performance between combining the algebra vector with the BGP feature vector and using them separately, but dissecting the two vectors will be the job of our machine learning model.

2.1.3 Data

For their experiment, they used 10000 random queries, taken from the USEWOD2016[1] challenge, and filtered for static data. We think that their database size is only barely enough to have a good approximation of real world queries. Since USEWOD contains around 43GB of text logs and an estimated 110 millions of queries, it is questionable if 10000 queries are enough to represent the real world nature of the USEWOD dataset. For this reason, we extended our dataset to 40000 random queries, but see a potential improvement in further increasing the database. The training size is also discussed at the end of the paper in Section 6. It is mentioned that it is very time consuming to build a bigger dataset and makes similar work hard to compare. We want to tackle this problem in this thesis: Make it possible to decrease the overall training time, which enables a bigger dataset and a more representative solution.

Their 10000 queries are run 11times: The first time 'cold', meaning the data is not cached, and then ten times in a 'warm' state. For their prediction of warm queries, they use the mean over those ten executions. This distinction solves one thing: it makes predicting cold queries easier. One of the most important features is whether or not a query has been run before and therefore stored in the cache. Rather than knowing or predicting beforehand whether data will be in cache, the dataset is split, and the model is trained on both datasets. However, this distinction is problematic for several reasons:

- A query is rarely entirely cold. Assuming a server has run a decent amount of queries, it will reuse interim results out of cache, which will significantly reduce the execution time. A distinction into 'partly-warm' and 'warm' would be a more accurate description for most queries.

- In the case of a server-side query scheduling application, this distinction is unusable. Knowing whether or not a query is in cache requires building a table of past queries. Accessing this table creates additional overhead, which is not wanted. A scheduling algorithm would therefore always need to use the cold model. We

[1] https://eprints.soton.ac.uk/385344/

need a satisfying performance on cold queries to make an efficient query scheduling possible.

- In the case of a front-end application, the accuracy is of less importance. The probability of a user testing the prediction versus performance is small.

We will adopt the distinction of cold and warm queries but hope to achieve a good enough performance, to only use cold queries in the future.

2.1.4 Evaluation Metric

As their cost function, they use relative error:

$$relativeerror = \frac{1}{N} \sum_{i=1}^{N} \frac{\mid actual_i - estimate_i \mid}{actual_{mean}} \tag{2.1}$$

We adopt this error function for our model but need to clarify $actual_{mean}$ further, which is discussed in section 3.2. While the usage and reasoning for $relativeerror$ are sound, they never compared their error to using no model. If we replace $estimate$ with $actual_{mean}$, we get a benchmark of using no model and using only statistics. Using such a reference point is important for two reasons:

- Since we take random queries from the dataset, we need to establish a baseline of our database. Otherwise, the comparison to previous work is unfair.
- In our database building process, we experienced some variance in our relative error over the database, since the query execution time is not only dependent on the query itself, but also on the server load and network connection. Without using a benchmark of the dataset, the improvement over old work could be partly based on happenstance.

2.2 Paper "A Machine Learning Approach to SPARQL Query Performance Prediction"

For the slightly older paper by [Hasan, 2014], we want to mainly highlight the differences to [Zhang et al., 2016].

2.2.1 Feature Modeling

In their feature extraction, they also construct an algebra feature vector with tuples and a GED vector. For their GED vector they first cluster queries using k-medoids [Kaufman, 1987], then calculate the GED from every query to the cluster center queries and transform this distance using:

$$sim(p_i, C(k)) = \frac{1}{1 + d(p_i, C(k))} \tag{2.2}$$

Where $d(p_i, C(k))$ is the GED between query graph p_i and cluster center query $C(k)$. This calculation is not necessary: The GED is mainly a feature to describe the structural properties of a query. To which graph we compare this to is irrelevant, as long as graphs are somewhat unique. For example, if we take five SELECT-queries the GED distances will obviously fail to provide an essential feature, but if we carefully choose unique queries beforehand to compare to we can refrain from any clustering algorithm.

2.2.2 Data

The dataset of queries is generated from the DBPSB[Morsey et al., 2011] benchmark[2] query templates, which cover the 25 most commonly used SPARQL query features on DBPedia[3]. They generate in total 2100 queries from the templates and split it up into training, validation and test set. Apart from the point we already made in section 2.1 regarding the database size, the query templates disregard any outliers:

> [Morsey et al., 2011] Applying BorderFlow to the input queries led to 12272 clusters, of which 24% contained only one node, hinting towards a long-tail distribution of query types. To generate the patterns used in the benchmark, we only considered clusters of size 5 and above.

The usage of generated queries is fine for presenting a more isolated case but makes it hard to argue for a real-world application.

2.2.3 Prediction

For predicting they achieve the best result with using both algebra and GED vector and the SVM algorithm with 15 clusters.

There are many more relevant features other than the query structure influencing query execution time. Processor load, caching state, network delay and others play a major role. That is why we want to approach the underlying problem a little differently: While they looked at queries more isolated, we want to leverage the query sequence in section 3.3 to get a more accurate approximation of the solution. So instead of independent queries, we feed a sequence into our model. Having a sequence, we hope to capture the state of the server and achieve better performance.

[2] http://aksw.org/Projects/DBPSB.html
[3] http://dbpedia.org

3

Implementation

3.1 Setup

We adopt a similar experiment setting of [Zhang et al., 2016] for comparability and as a reference point for improvement. Our dataset consists of 40'000 random queries, taken from the USEWOD2016 challenge, which provides query logs from a DBPedia3.9[1] endpoint.

The queries were executed 11 times. We use the first one as a "cold query" execution, since there is no cache present beforehand. We use the average of the following 10 queries as "warm query" execution, since some data is presumably cached from the cold query. A random 20% of our dataset signifies our test set and 80% our training set. The maximum execution time on the server was set to 30 seconds, but any existing timeout in the provided log remained. In our dataset build, we observed high variance in execution times, which signifies the importance of providing the relative mean error of the dataset itself, as a benchmark.

Our server instance hosted Virtuoso 7.2[2] on a Debian 3.16 machine with 64GB RAM and 24CPUs at 2.2GHz. We followed Virtuoso's performance tuning guideline[3] by setting following parameters:

```
1 NumberOfBuffers=5450000
2 MaxDirtyBuffers=4000000
```

Listing 3.1: Virtuoso settings

The TensorFlow models were trained on an iMac (2013) with 32GB RAM and i7-4771 CPU @ 3.50GHz. All models were trained using TensorFlow v1.1.

[1] http://wiki.dbpedia.org/services-resources/datasets/data-set-39
[2] https://virtuoso.openlinksw.com/
[3] https://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtRDFPerformanceTuning

3.1.1 Feature Modeling

To use our queries with machine learning models, we first have to transform them into vectors. Each value in the vector is a feature of the current query. The performance of our models highly depends on how well our feature vectors can describe our queries.

Algebra Vector

Our first vector tries to capture the syntactical and structural information in the query. Meaning that different SPARQL operators have a different influence on the performance, so if we can parse this structural information, we can extract meaningful features. Additionally, the syntax of the query decides in which order the query is supposed to be executed. We can parse this syntax into a graph and leverage the graph structure to extract further features.

To build our algebra vector, the query is first parsed using Apache Jena-3.2[4], which provides an algebra tree. We then parse this tree for features. We extract the height of the tree, and a triple for every SPARQL command, containing the occurrence of the operator, the maximum and minimum height. This results in a set of tuples for every SPARQL operator: $\{(c_i, minh_i, maxh_i)\}$, where c_i is the operator's count, $minh_i$ the operator's minimum height in the algebra tree and $maxh_i$ the maximum height. These tuples are all concatenated in the end. Additionally, we add a representation of the amount and type of the occurring triple patterns. This procedure results in a vector of 52 features.

GED Vector

In our second vector we further extract features from our query graph structure. The graph edit distance (GED) measures the minimum amount of graph operations to transform one graph into another. Such operations include insertion, deletion of isolated vertex or edge, and changing the label of a vertex or edge. However, this problem is NP-complete.

The Graph Edit Toolkit [Riesen et al., 2013] provides implemented approximation algorithms that run in polynomial time, such as the Beam algorithm. According to [Riesen et al., 2013], this exaggerates GED, but in our application accuracy is not highly relevant, as we only want to differentiate queries from another. As long as we have a consistent error, this is not of concern. In our implementation, we chose to use the Jonker-Volgenant (JV) algorithm, originally proposed in [Jonker and Volgenant, 1987], mainly because of its slight speed advantage over Beam. However, even with its stable performance, relying on an algorithm with cubic running time is impractical in most scenarios. It would be a meaningful improvement if we can achieve comparable performance to state of the art without any GED.

[4]https://jena.apache.org/

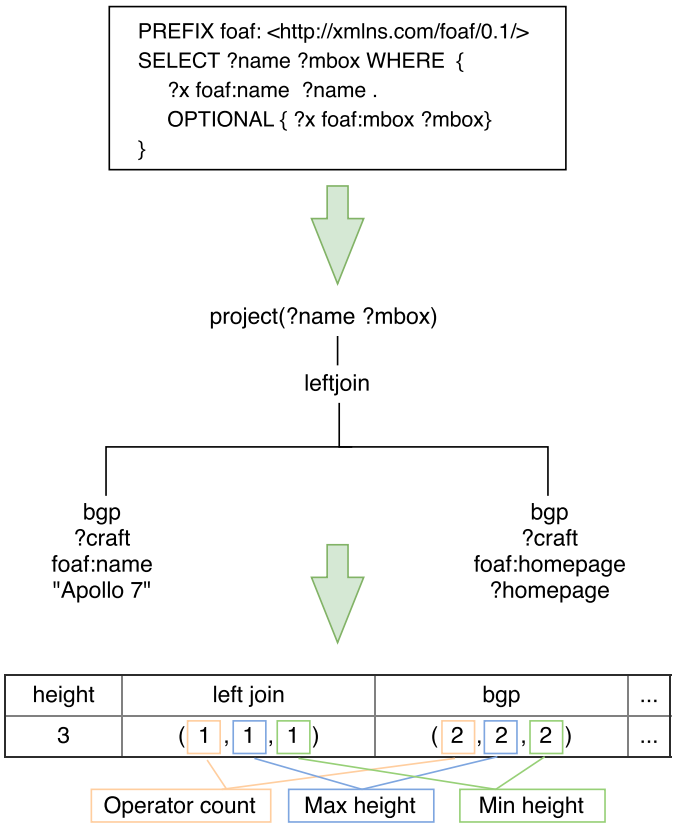

Figure 3.1: Algebra Feature on example query

For every query, we record the GED to 10 benchmark queries, taken from DBPSB[Morsey et al., 2011], mentioned in subsection 2.2.2. Using eight queries less than in [Zhang et al., 2016] should provide the first step towards a meaningful speed improvement.

3.2 Multilayer Perceptron

3.2.1 Setup

We chose first to build a working model and then optimize from there. Therefore our parameters are all within a reasonable boundary, but arbitrary in this first step. We chose to use one hidden layer with size 1024. The hidden layer has a rectified linear unit (ReLu) as the activation function and a dropout layer[Srivastava et al., 2014] with a dropout-probability set at 20%. The model's optimizer is TensorFlow's implementation of Adam Optimizer[Kingma and Ba, 2014] with a learning rate of 10^{-3}.

Evaluation Metric

We adopted the cost function of [Zhang et al., 2016], namely *relativeerror* (2.1), but need to clarify one distinction. In our training dataset $actual_{mean}$ is defined over the training dataset and in the test set accordingly.

As a benchmark for our evaluation, we set *prediction* to $actual_{mean}$ over the test set, which is equal to using no model and relying on query statistics alone. Additionally, we added State of the art error, which refers to the best score [Zhang et al., 2016] achieved using their two-step prediction method. The model gets evaluated over 100 epochs, meaning after 100 forward an backwardpasses of all training examples, which seems to be enough for the network to stabilize and converge.

In the following, we will optimize on cold queries as a default unless otherwise noted, as this is the more difficult task. In the end, we will again make the comparison between the cold and warm queries.

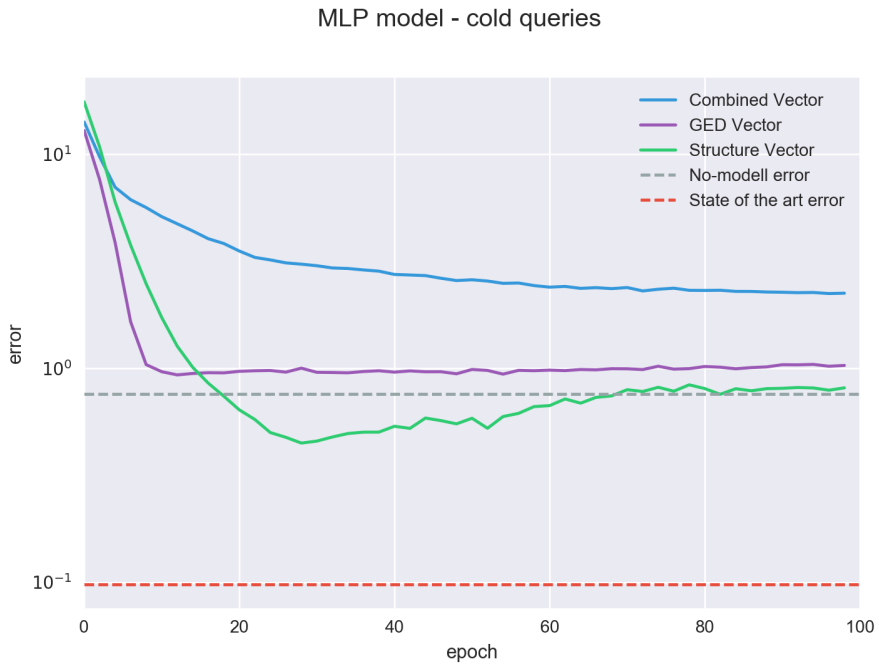

Figure 3.2: Multilayer Perceptron performance

First Model

In our first, unoptimized version of the model (Figure 3.2) we see that the GED vector is converging the fastest, and the algebra vector has overall the best performance. We can observe that errors start all very high. Also, the error of the algebra vector is

increasing after finding local minimum at around 0.8%, which can be an indication of overfitting.

3.2.2 Model Optimization

Weight Initialization

One approach to optimization is to adjust variable initialization. How to initialize weights in an optimal fashion is a topic of research, but there exist several best practices and known practices to avoid.

We want to avoid setting all weights to the same value, to break symmetry. Meaning that if we set every weight to one, each hidden unit will get the same signal, which is the sum of inputs multiplied by the corresponding weights. Even worse, if we set all weights to zero, all hidden units will get zero signal. Therefore we want to initialize weights randomly, or as a minimum with different values.

Before, our weights of the hidden layer are initialized with the default settings for TensorFlow's `tf.random_normal`:

```
new_layer = {'weights': tf.Variable(tf.random_normal(
               [layer_config[i-1], layer_config[i]], mean=0, stddev=1)),
             'biases': tf.Variable(tf.random_normal(
               [layer_config[i]], mean=0, stddev=1))}
```

Listing 3.2: Weight initialization

Which means that it in our first model we initialized the weights with a mean at zero and standard deviation at 1, which is not ideal in our case for two reasons:

- Because we use a ReLu activation, it is a good practice to use a slight positive bias to avoid "dead neurons".
- The weights seem too large for our case: The majority of our labels are around 0.06, setting the standard deviation to +/- 1 is too high. However, if we set the weights too small, then the signal shrinks as it passes through the layers and may end up too small to be useful. It may be overengineering for our model, but initializing the weights correctly is essential to getting a good result.

One of the most famous weight initialization methods is *Xavier Initialization*, proposed by [Glorot and Bengio, 2010]. It defines the variance at:

$$Var(W) = \frac{6}{n_{in} + n_{out}} \tag{3.1}$$

n_{in} and n_{out} are the numbers of inputs and outputs of the layer.

[He et al., 2015] found that using Xavier initialization in deep neural networks with 30 layers and ReLu activations does not lead to convergence and suggest a different variance:

$$Var(W) = \frac{2}{n_{in}} \tag{3.2}$$

Note that in both versions the variance of the bias initialization and the mean of weight initialization are both set at 0.

To see which weight initialization is better in our case we run our MLP model with the GED vector and two different configurations: First, with one hidden layer of size 128. Second with five hidden layers with size 64. We chose these settings to find out whether the difference is noticeable in a deeper network, as [He et al., 2015] describes. Both initialization methods are run five times and are averaged. We also include our original setting of standard deviation at 1 for comparison.

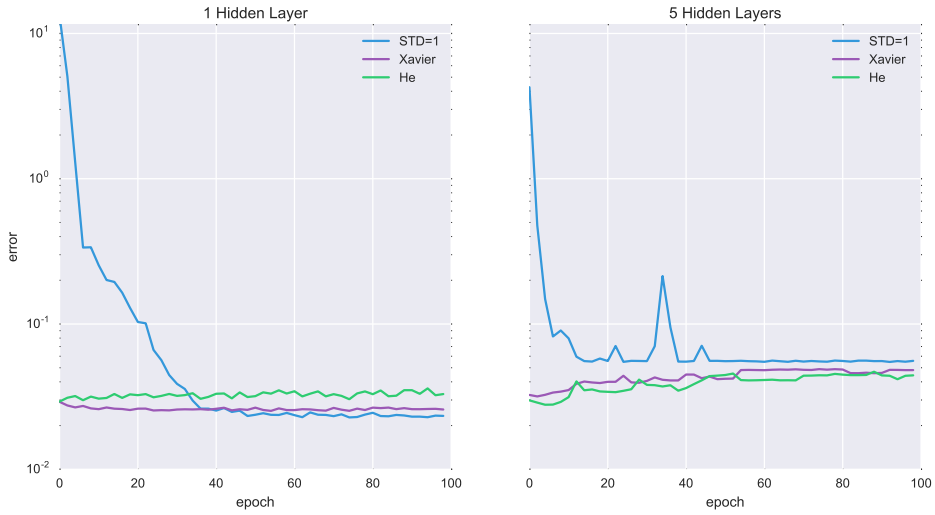

Figure 3.3: Comparison between Xavier and He initialization

In Figure 3.3 we can see that with one hidden layer Xavier is a little less prone to overfitting than He initialization. With five hidden layers, we can observe the opposite. The effect of He initialization performing better is likely to increase as the network gets deeper but is subject to future work. As our networks are not very deep, we chose Xavier initialization.

Early Stopping

Until now, we evaluated our model at epoch 100. However, this might be too late for models that are prone to overfit. In Figure 3.2 we can observe that the algebra vector

would have performed better if we chose to stop at around epoch 25. Early Stopping refers to an approach in training models to avoid overfitting. As the name suggests: It tries to avoid overfitting by stopping early at an inflection point, where the error starts to increase. There are many ways to approach stopping early. [Prechelt, 1998] explore three classes of stopping criteria and conclude on three selection rules for choosing a stopping criterion. We adopt Rule 2, which says:

> 2. To maximize the probability of finding a "good" solution (as opposed to maximizing the average quality of solutions), use a GL criterion.

A GL criterion is describing the generalization loss, which describes the relative increase of validation error over the minimum-so-far (in percent). We slightly modify to use the test error instead of the validation error:

$$GL(t) = 100 * (\frac{E_{te}(t)}{E_{opt}(t)} - 1) \tag{3.3}$$

Where $E_{opt}(t)$ is defined as:

$$E_{opt}(t) := \min_{t' \leq t} E_{te}(t') \tag{3.4}$$

Meaning $E_{opt}(t)$ is the lowest test error in the epochs before t and $E_{te}(t)$ is the test error at epoch t. This stops the training process at epoch t with $GL(t) > x$. The difficulty is to define an appropriate value for x: If set too low, we do not allow for an error adjustment out of a local minimum, and if set too high we will not stop early before overfitting. We include this new parameter into our parameter optimization.

Additionally, we add a *patience* variable, which signifies how many epochs to wait until early stopping is considered. If we would consider early stopping at the very beginning, a first misstep by the optimizer could trigger an early stop, which we want to prevent. In our case, we want to set *patience* at 10.

Parameter Optimization

The model is still prone to overfitting, as the performance still varies from iteration to iteration. The tendency to overfit is mostly due to a lack of hyperparameter optimization. We still define the depth and size of the network, batch size, learning rate, dropout probability all by hand.

One approach to optimizing these parameters is Bayesian optimization, which is a sequential model-based optimization algorithm. Intuitively the algorithm uses previously computed points as input to suggest the next set of parameters, for the objective function, which will likely give a better result. The objective function is in our case to minimize the error rate when using the test set, averaged over the last ten epochs. This optimization process is computationally very expensive, which is why the model will only be evaluated over 80 epochs.

To make a custom implementation of Bayesian optimization would go beyond the scope of this work. Fortunately, there is an excellent implementation by the Harvard Intelligent Probabilistic System Group[5], called "Spearmint"[6], which is originally based on the work by [Snoek et al., 2012]. Spearmint provides a simple wrapper to a python function to do parameter optimization. Doing this optimization is computationally very expensive, which is why we have narrowed down our parameters to following scopes:

```
0.0001<learning_rate<0.1
8<batch_size<256
0<num_network_layers<4
8<layer_size<256
10<early_stop<100
```

Listing 3.3: Optimization Parameters

We ran the optimizer for 250 iterations on our vectors with following results:

Vector	Minimum calculated error	Minimum observed error
GED	2.454% (+/- 0.163)	1.9957%
Algebra	4.534% (+/- 1.383)	3.7436%
Combining GED&Algebra	5.281% (+/- 22.171)	2.0731%
[Zhang et al., 2016]	-	11.06%

Table 3.1: Bayesian Optimization result.

We see that the GED vector outperforms the algebra vector, but both are well under the current state of the art. Additionally, the combined vector's range of error suggests we still experience overfitting and could benefit from a manual correction of the early_stop flag.

When using these optimized parameters, the results were not always consistent. At a few rare iterations, the model overfitted completely and got stopped out. We think that this is mostly caused by the randomness in weight initialization and unavoidable. Also, since we can simply save the current model and reuse it, being able to reproduce the performance from the start is not a prerequisite. Another possible influence could be that we split our training and test set randomly before every run, so if we predominantly test on outliers, the error will be higher.

We can see in Figure 3.4 that our MLP outperforms state of the art with every vector combination. It appears that the GED vector performs significantly better than the other two options. Also, it is interesting that the combined vector performs almost identical

[5]http://hips.seas.harvard.edu
[6]https://github.com/HIPS/Spearmint

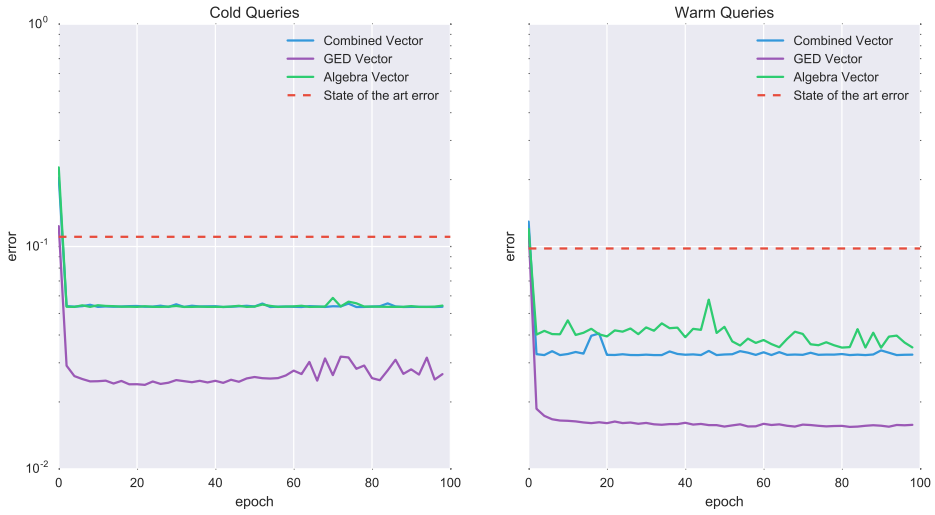

Figure 3.4: MLP with implemented optimization.

to the algebra vector with cold queries. Note that we only optimized our models for cold queries and used the same settings for warm queries, which very well be not ideal. Final results, averaged over the last ten epochs are as follows:

Vector	Cold Error	Warm Error
GED	2.76%	1.57%
Algebra	5.38%	3.73%
Combining GED&Algebra	5.35%	3.3%
[Zhang et al., 2016]	9.81%	11.06%

Table 3.2: Final result comparison using our MLP model.

3.3 LSTM Network

3.3.1 Idea

One disadvantage of MLPs is their inability to include the history of inputs. While a MLP can only map from input to output vectors, and therefore considers all samples independently, a recurrent neural network (RNN) can, in principle map the entire history of previous inputs to each output. The idea is to provide a memory of previous inputs that persists as an internal state and can influence the network's output. However, RNN has its limits when trying to map long-term dependencies. Theoretically, it

is possible for an RNN to learn long-term dependencies by carefully optimizing its parameters, but in practice, RNN seems unable to learn them. This problem was explored by [Bengio et al., 1994], who found some reasons why long-term dependencies present a problem for RNN.

Introduced by [Hochreiter and Schmidhuber, 1997], LSTM is a special kind of RNN network, that are built to avoid this long-term dependency problem. The rough idea is to add a cell state, to which the LSTM cell can add or remove information. This modification of information is regulated by gates, which are defined by the type of cell used.

3.3.2 Challenge

Besides being an excellent exercise to building an LSTM network, we originally hoped for it to solve one problem: The network should learn whether a query is taken from cache or not. As discussed in chapter 2, we think splitting the database into cold and warm is unpractical and unrealistic. However, it is crucial to know whether or a query is already in cache. Unfortunately, this introduces several new challenges. Some of them are:

- Depending on the log, the network maybe has not enough data to learn such a feature.
- The state of the cache depends on Virtuoso's software: Just because the query uses somewhat similar data, it is not guaranteed to get it from the cache. Of course, when querying the same data, this is not a problem.
- We would need to capture the used data as a feature. Currently, we only capture the structure of the query. To do so, we have to extend our feature extraction and find a way to vectorize the data a query tries to access.

3.3.3 Model

One difficulty of building the network was the formatting of the data: To learn from long-term dependencies, we need to feed in the data as a sequence. This sequence gets too long in our case, as we want to learn from all past queries. On the other hand, if we define the sequence too short we might miss an identical past query.

The solution to this was a persistent state: Normally the hidden state would get reset after every sequence, but we can "inject" the last hidden state of the previous sequence into the next sequence to learn from all past queries that were fed into the network. However, the persistent state has to be reset at the beginning of every epoch. Otherwise, the network would think that every query is in the cache after epoch 1. This reset leads to spikes in error when plotting the performance of our network, as at

the beginning of every epoch the hidden state is set to zeroes. Without this necessary modification to the model, we cannot achieve any meaningful result with LSTM.

Setup

In our LSTM network, we use the newly developed NASCell [Zoph and Le, 2016], three layers of cells, 100 epochs, the already mentioned Adam Optimizer [Kingma and Ba, 2014] with a learning rate of 10^{-3} and a batch size of 16. The model also uses the Xavier-initializer [Glorot and Bengio, 2010], mentioned in section 3.2.2.

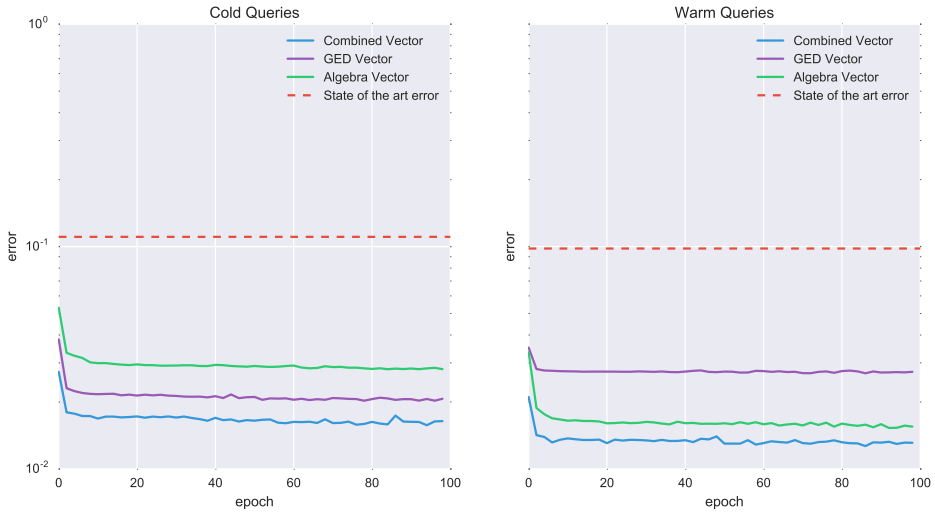

Figure 3.5: LSTM performance

We observe in Figure 3.5 that the performance is very good and does not need further optimization. This model outperforms our MLP in almost all vector combinations. It is also nice to see such a good performance using only the algebra vector because with this approach we can save computation time in our feature extraction. Final results, averaged over the last ten epochs are as follows:

Vector	Cold Error	Warm Error
GED	2.05%	2.72%
Algebra	2.83%	1.55%
Combining GED&Algebra	1.62%	1.31%
[Zhang et al., 2016]	11.06%	9.81%

Table 3.3: Final result comparison using our LSTM model.

3.4 Distributed TensorFlow

3.4.1 Code Adjustments

As a model grows, one experiences quickly that TensorFlow needs a lot of computing power. With minimal code adjustment, TensorFlow can be executed in a distributed fashion. It is important to know the difference on how the execution of the graph takes place. According to the docs[7]:

> A TensorFlow "cluster" is a set of "tasks" that participate in the distributed execution of a TensorFlow graph. Each task is associated with a TensorFlow "server", which contains a "master" that can be used to create sessions, and a "worker" that executes operations in the graph.

To coordinate between the nodes in a cluster, we mainly need to define two variables:

ClusterSpec, a dictionary that describes all tasks in the cluster. It maps job names to a list of network addresses. For example:

```
tf.train.ClusterSpec({
  "worker": [
      "claudio09:2222",
      "claudio10:2222",
  ],
  "ps": [
      "claudio11:2222"
  ]})
```

Which results in 3 available tasks:

```
/job:worker/task:0
/job:worker/task:1
/job:ps/task:0
```

Every node gets the same ClusterSpec.

Additionally, every node has a Server variable, which contains a set of connections to other tasks, the node's job_name, and its task_index. Obviously, every node gets a different instance of tf.train.Server. As we can see above, jobs are split between PS and worker. The difference is that Parameter Servers (PS or parameter devices) hold the variables, and are updating variables at the appropriate time. A worker is an assigned the more computationally intensive part of the model, like pre-processing, loss calculation, backpropagation, and sends updates to the PS when available. Figure 3.6 is illustrating the communication further.

[7] https://www.tensorflow.org/deploy/distributed

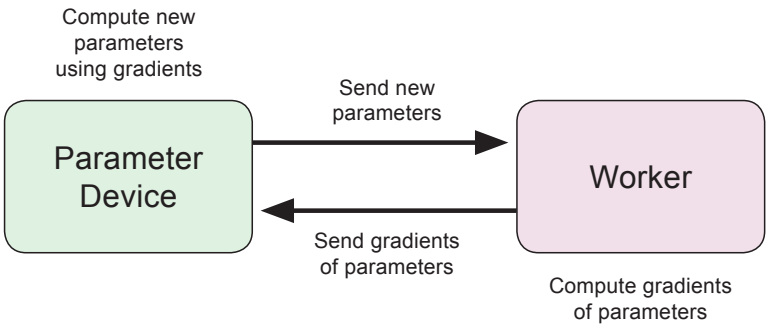

Figure 3.6: PS Interaction with Worker

3.4.2 Cluster Design

With that, every node knows how to communicate with each other. Unfortunately, TensorFlow lacks support for many cluster managers, meaning that when using the IFI cluster, that uses Slurm[8] as a cluster manager, one would have to define the ClusterSpec and `tf.train.Server` manually. There is an unsupported slurm manager for TensorFlow on github[9], that we successfully deployed. When using the slurm manager for TensorFlow, the ClusterSpec gets automatically built using available nodes.

There are several design choices when deploying a model in a distributed fashion. We have to decide between "in-graph replication" and "between-graph replication" and between "asynchronous training" and "synchronous training".

- Synchronous Training: All workers are coordinated. After reading the parameters and computing the gradient, the worker waits for others to finish. When all workers are done with the current iteration, a PS averages the gradients and does an update based on the average. Synchronous training is only as fast as the weakest link: if one worker is slow, every other worker does nothing until the slow worker finished (Figure 3.7).

- Asynchronous training: Workers can update each other's work freely, without any locking mechanism (Figure 3.8). Each worker is training the model as if it were isolated, but parameters are shared with other workers.

- In the case of "in-graph replication", one client contains the parameters and assigns intensive computation to workers, similar to a resource manager. The problem with

[8] https://slurm.schedmd.com/
[9] https://github.com/jhollowayj/tensorflow_slurm_manager/blob/master/slurm_manager.py

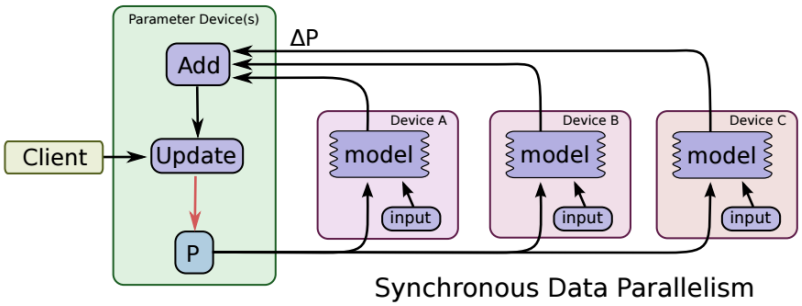

Figure 3.7: Synchronous Data Parallelism [Abadi et al., 2016]

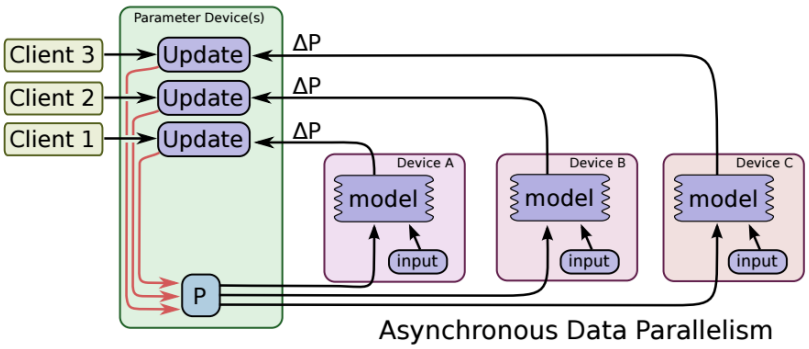

Figure 3.8: Asynchronous Data Parallelism [Abadi et al., 2016]

this replication is that it is not scalable enough[10], when the model has hundreds of clients, the "master client" distributing the workload, becomes a bottleneck.

- With "between-graph replication" each worker uses a separate TensorFlow process and builds a separate Graph. This makes synchronous training difficult because one has to synchronize different training loops.

While synchronous training achieves a better accuracy, asynchronous training has a higher throughput [Abadi et al., 2016]. Between-graph replication with asynchronous training is probably the most common choice as of now, and our choice to build a test model.

3.4.3 IFI Cluster Performance

To measure the performance of IFI's Kraken cluster, we start by looking at the throughput. We modified our LSTM model in section 3.3 so that it can be deployed distributedly. We measure the throughput in queries per second and compare it between

[10] http://stackoverflow.com/questions/39658422/tensorflow-in-graph-replication-example

different configurations of PS and worker tasks. We included the throughputs of running the model in a non-distributed fashion on an iMac and one Kraken node.

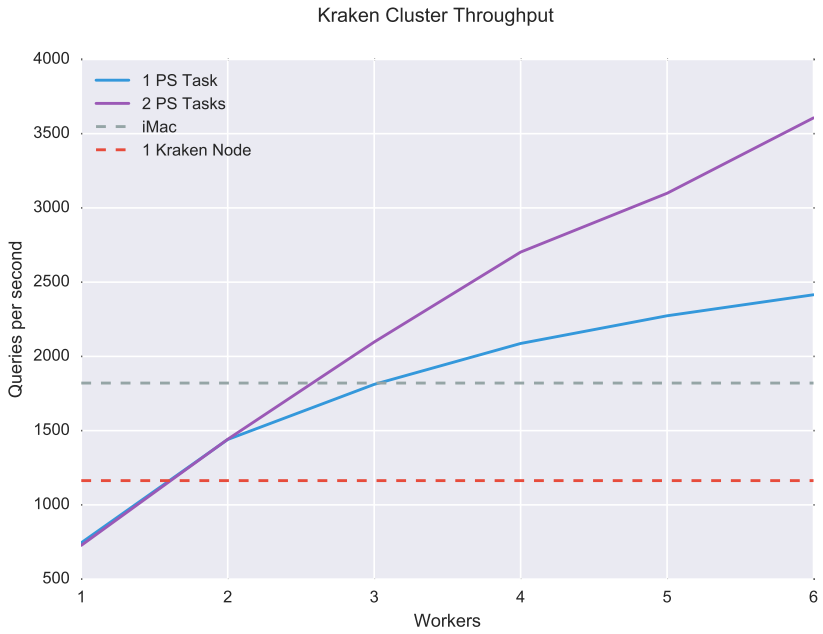

Figure 3.9: Kraken Cluster Throughput

We can observe in Figure 3.9 that the throughput scales sub-linearly. This was to be expected, due to the additional network overhead. It is interesting to see that using only one PS task leads to a faster convergence to a maximum throughput, most likely due to the PS task becoming the bottleneck. When using two PS tasks, this effect is delayed. Unfortunately, as we only have limited number of nodes, this effect is difficult to study at a larger scale.

This supports the original observation in [Dean et al., 2012], where they saw a sub-linear training speed-up for four different models using DistBelief, the predecessor of TensorFlow.

The performance of the Kraken cluster is rather underwhelming. We think the problem is two-fold: First, TensorFlow is mainly intended to be deployed on GPUs. Second, the CPU base clock is rather low, which could explain the difference to the iMac's performance. However, our models were only optimized for the result, and not speed. For future work, we suggest looking into potential bottlenecks of the cluster. One potential bottleneck is the data feed: TensorFlow supports queues[11], where it is possible to preprocess the data and not do a direct feed with python variables using `feed_dict`. The usage of queues could shorten the time where the CPU or GPU is idle.

[11] https://www.tensorflow.org/programmers_guide/reading_data

Asynchronous Training

As previously mentioned in asynchronous training we only have one shared copy of each variable. Each worker sends the gradients to its PS task and applies it to the appropriate variable, which is essentially a running aggregate of updates received. The updates may be done independently from each worker.

The notion of only having one copy of variables raises interesting questions. Intuitively we want to compare the different results of our workers to get the best result. We would have to compare if the higher throughput, thanks to asynchronous training, leads to a shorter time to accuracy.

[Chen et al., 2016] looks at this performance difference, and finds that with modern GPUs and a fast interconnect a synchronous model runs acceptably fast. With our environment at IFI this is likely not applicable.

4

TensorFlow

4.1 Session and Graph

We wanted to explain two characteristics of TensorFlow, that can be confusing when first reading TensorFlow code. On a very high level, every model consists of two parts: a TensorFlow Graph and Session.

A graph defines the computation. It does not hold values or compute any operation but defines the operations specified in the code. A session, on the other hand, executes the graph, holds the actual values, results, and variables. According to the documentation[1]:

> A Session object encapsulates the environment in which Operation objects are executed, and Tensor objects are evaluated.

Therefore if we define a var like so:

```
graph=tensorflow.Graph()
with graph.as_default():
  var=tensorflow.Variable(1)
```

Listing 4.1: Variable initialization

The value of tf.Variable is not stored in the python variable var. It is only a reference to the graph. TensorFlow creates a default graph, which means the first two lines are redundant in this example.

To access the value of var we have to add the following:

```
with tensorflow.Session(graph=graph) as sess:
  sess.run(tensorflow.global_variables_initializer())
  #global_variables_initializer() returns op
  #that initializes global variables
  print(sess.run(var))
  #1
```

Listing 4.2: Session initialization

This provides the basic idea to building a TensorFlow model: We first built the graph (in this case only consisting of variable var) and executed the graph in a session. Obviously, this construct is very basic but should give a good intuition to reading TensorFlow code.

[1] https://www.tensorflow.org/api_docs/python/tf/Session/

4.2 TensorBoard

As developing machine learning models can be confusing and very complex there is a need for visualization tools. TensorBoard[2] is a suite of visualization tools that deserves a highlight since it eases the debugging and understanding of one's models. TensorBoard is included in the normal TensorFlow distribution and gets started with:

```
tensorboard --logdir=path/to/log-directory
```

If we would like to learn how the error rate is varying over time, we can simply attach `tf.summary.scalar`[3] to the node. Another useful application of TensorBoard is the ability to visualize the graph. This can help to understand any architectural misconfiguration of the network when debugging the performance. Figure 4.1 shows our LSTM network visualized with this tool.

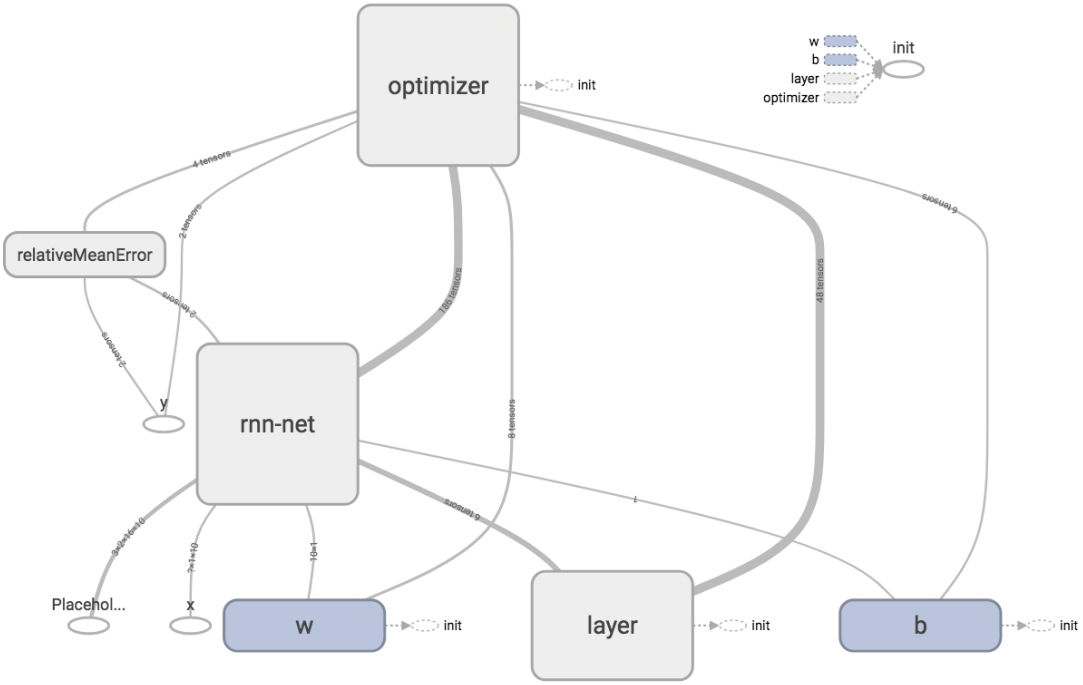

Figure 4.1: LSTM graph

One disadvantage is that TensorBoard does not support replicated summary writer, meaning that in a distributed setting TensorBoard will not work as expected. However, as every task in a distributed model is identifiable with its `task_id` and `task_name` we can filter for a single node and have it writing any scalar for TensorBoard, identical to using TensorBoard with a single machine.

[2] https://www.tensorflow.org/get_started/summaries_and_tensorboard
[3] https://www.tensorflow.org/api_docs/python/tf/summary/scalar

5

Critical reflection

It is hard to give a definitive solution to a problem using machine learning. We can set a time limit on training time, or experimentally look for a convergence and try to explain the reasoning behind the solution, but at the end, we lack a definitive answer. Maybe we experience a local minimum in our problem and would get a much better solution if we ran the training longer. Maybe we missed a better solution, due to a bad setting in model parameters. Machine Learning is still very much a "black-box" approach, that achieves empirically very impressive results, but lacks in fundamentally proven solutions and ways to visualize the reasoning. It is a fascinating and rapidly changing field, but if one lacks experience in practical machine learning, developing and optimizing a model can easily get confusing due to the vast amount of options that exist.

Using TensorFlow at this time has many advantages, but also one disadvantage: It is still very actively being developed. During this project, they released V1.0, which had breaking changes due to many refactorings. The version change made any tutorial or preexisting code snippets almost unusable. Of course, it forced a deeper dive into TensorFlow and understanding the underlying mechanism, but the resulting workload was rather high.

5.1 Comparison with related work

This is a summary of our ongoing comparisons, mentioned in chapter 3. We have found multiple configurations that work better than the current state of the art [Zhang et al., 2016]. First, we found that both our models outperform state of the art without using GED:

Model	Relative Error (Cold)	Relative Error (Warm)
MLP(Algebra)	5.38%	3.73%
LSTM(Algebra)	2.83%	1.55%
[Zhang et al., 2016]	11.06%	9.81%

Table 5.1: Performances using only Algebra Vector.

This saves computation time when preparing the query. This is crucial in a practical application, as the delay of a prediction only worsens the usability of the model. There is still room for improvement, but not relying on GED is the clear way to go for future work.

Additionally, we want to highlight the best performance we achieved versus state of the art, which is more than a six-fold improvement:

Model	Relative Error (Cold)	Relative Error (Warm)
LSTM(GED+Algebra)	1.62%	1.31%
[Zhang et al., 2016]	11.06%	9.81%

Table 5.2: Best Performance we achieved overall.

5.2 Discussion of open issues

Similar to every other machine learning model and case study, there are more good practices than fundamentally proven ways to justify design decisions. Therefore, we can often only give empirical observations.

One issue we had was that it is difficult to generalize a solution while having a vast amount of variables to define. For example, the performance of a query relies on the network connection of the client and server, the server's workload, the server's free RAM, Virtuoso's query engine, Virtuoso's cache state and many more factors. These conditions make it hard to reason why a query performs the way it does. We also saw high variability in query execution times, independent of hardware used. Running the same queries twice, without any change in configuration did result in over 10% difference in relative mean error over the dataset (both warm and cold queries). There might be some gain in setting up Virtuoso with a VirtualBox[1], where we have more control over the system's resources.

In our problem, the time to prediction is clearly important. However, there's no good way to measure this. In [Zhang et al., 2016] they measure *Time1k*, which means the time it takes for their model to extract features and predict the time for 1000 queries. This is highly hardware dependent and requires us to rebuild their work to have a fair comparison. Additionally, the time to train a model is not highly relevant, if you have the option to reuse pre-trained models. In the end, we are interested in the time it takes to extract features and for the model to make a prediction.

[1] https://www.virtualbox.org/

In our models, we have two factors, that are to some degree random: the weight initialization and the split between training and test samples (fixed ratio). This introduces some variance in our results that make it difficult to generalize a definitive solution in the end.

6

Future Work and Summary

6.1 Future Work

Optimizer: For both of our models, we used Adam Optimizer, without testing other options thoroughly. This could be a possible route for improvement. Better yet, a recent paper by [Andrychowicz et al., 2016] showed the potential of using a meta-learner: Instead of a handcrafted optimizer that trains our model, we define an RNN that gets trained on our gradients and suggests the next step.

Dynamic Data: In this model, we only considered static data. With dynamic queries, the data gets updated. We should investigate the performance of our model on dynamic data and what features or different techniques are appropriate for the use case.

Training Size: We used a bigger dataset than previous work, but the model would benefit from seeing more data. By seeing more diverse data, the model will get a more robust prediction performance.

LSTM Optimization: Our LSTM ended up being much more time-consuming to train than the MLP. That is the reason why we only looked at Bayesian Optimization for our MLP model since in the optimization process the model has to be executed numerous times. In the future we will look at using Spearmint in our cluster, and how to approach distributed Bayesian Optimization.

Different Dataset: All work on predicting SPARQL query performance relies on DBPedia at the moment. In the future, we will look at different datasets, for example, LUBM[1], and see how the performance compares. It would also be interesting to see how well our model performs if we mix our queries from different datasets. Since we only use syntactic features, we might be able to generalize a broader solution.

Distributed TensorFlow: We saw that there is only few research done in comparing the different distributed TensorFlow models. There is a wide variety of hardware setups, and it would be valuable to generalize some reasoning behind different design choices.

[1] http://swat.cse.lehigh.edu/projects/lubm/

Feature Modeling: In our approach, we only used the syntactic and structural features of a query. In the future we want to explore feature extraction of data, that was queried in the past. One possible route is to vectorize DBPedia using word2vec and leverage the distance between vectors of queries as features.

6.2 Summary

In this thesis, we studied existing work on predicting SPARQL query performance. In our implementation, we adopted existing feature modeling and trained machine learning models with it. We explored two different TensorFlow models and ways to optimize their performance. We presented several models that outperform state of the art and save time in query feature modeling. Having a better performance with a reduced vector size, we identified that for our problem model performance is not the only goal, but rather to achieve a good balance between performance and time to prediction, which we will explore further in the future.

Additionally, we deployed TensorFlow on the IFI cluster and measured its throughput. We showed several design options in building distributed models and offer a potential reasoning for its performance.

References

[Abadi et al., 2016] 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 *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*.

[Andrychowicz et al., 2016] Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989.

[Bengio et al., 1994] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166.

[Chen et al., 2016] Chen, J., Monga, R., Bengio, S., and Jozefowicz, R. (2016). Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*.

[Dean et al., 2012] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. (2012). Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256.

[Hasan, 2014] Hasan, R. (2014). Predicting sparql query performance and explaining linked data. In *European Semantic Web Conference*, pages 795–805. Springer.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Jonker and Volgenant, 1987] Jonker, R. and Volgenant, A. (1987). A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340.

[Kaufman, 1987] Kaufman, L. (1987). Clustering by means of medoids. *Statistical data analysis based on the L1-norm and related methods*.

[Kingma and Ba, 2014] Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Morsey et al., 2011] Morsey, M., Lehmann, J., Auer, S., and Ngonga Ngomo, A.-C. (2011). Dbpedia sparql benchmark–performance assessment with real queries on real data. *The Semantic Web–ISWC 2011*, pages 454–469.

[Prechelt, 1998] Prechelt, L. (1998). Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer.

[Riesen et al., 2013] Riesen, K., Emmenegger, S., and Bunke, H. (2013). A novel software toolkit for graph edit distance computation. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 142–151. Springer.

[Snoek et al., 2012] Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

[Srivastava et al., 2014] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

[Zhang et al., 2016] Zhang, W. E., Sheng, Q. Z., Taylor, K., Qin, Y., and Yao, L. (2016). Learning-based sparql query performance prediction. In *International Conference on Web Information Systems Engineering*, pages 313–327. Springer.

[Zoph and Le, 2016] Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.

A

Appendix

A.1 Code

All of our code for this project can be found at https://github.com/derdav3/tf-sparql

List of Figures

List of Tables