A Neural Networks Approach to SPARQL Query Performance Prediction*

Daniel Arturo Casal Amat
Universidad Técnica Federico Santa María, USM
Valparaíso, Chile
daniel.casals@sansano.usm.cl

Carlos Buil-Aranda
Universidad Técnica Federico Santa María USM
Valparaíso, Chile
carlos.buil@sansano.usm.cl

Carlos Valle
Universidad de Playa Ancha UPLA
Valparaíso, Chile
carlos.valle@upla.cl

***Abstract*—The SPARQL query language is the standard for querying RDF data and has been implemented in a wide variety of engines. These engines support hundreds of public endpoints on the Web which receive thousands of queries daily. In many cases these endpoints struggle when evaluating complex queries or when receive they too many concurrently. They struggle mostly since some of these queries need large amounts of resources to be processed. All these engines have an internal query optimizer that proposes a supposedly optimal query execution plan, however this is a hard task since there may be thousands of possible query plans to consider and the optimizer may not chose the best one. Herein we propose the use of machine learning techniques to help in finding the best query plan for a given query fast, and thus improve the SPARQL servers' performance. We base such optimization in modeling SPARQL queries based on on their complexity, operators used within the queries and data accessed, among others. In this work we propose the use of Dense Neural Networks to improve such SPARQL query processing times. Herein we present the general architecture of a neural network for optimizing SPARQL queries and the results over a synthetic benchmark and real world queries. We show that the use of Dense Neural Networks improve the performance of the Nu-SVR approach in about 50% in performance. We also contribute to the community with a dataset of 19,000 queries.**

I. Introduction

The SPARQL query language is the standard for querying RDF data and has been implemented in a wide variety of engines (e.g., [12, 1, 26]). These engines support hundreds of public endpoints on the Web [9] which receive thousands of queries daily [8]. Though current SPARQL implementations work well for processing large workloads of relatively simple queries [23], in many cases these endpoints struggle when evaluating complex queries or when receive they too many concurrently [9].

They struggle mostly since some of these queries [8] need large amounts of resources to be processed. For processing queries, engines first need to "understand" these queries, which involve a complex process of scanning the data from disk (if it not in memory), process the operation within the query and return results to the user. It is well known that processing intermediate amount of results requires large amounts memory, and that may be problematic for most engines [19]. To improve the query plan execution all these engines have an internal query optimizer that proposes a supposedly optimal query execution plan. Query plan selection is a hard task since there may be thousands of possible query plans to consider and the engine may not chose the best one [22]. The development of a good query planner is considered the "Holy Grail" in database management systems. Existing DBMSs implement a key step of cardinality estimation by making simplifying assumptions about the data (e.g., inclusion principle, uniformity or independence as- sumptions) [22].

RDF systems rely on classical relational database systems optimizer [30] plus several heuristics specific to RDF [14]. However, these are still far from obtaining the best query plan due to the large amount of possible query plans for RDF, since in these databases there is no clear schema (and thus useful information about the data) that helps in the query processing process. To solve that issue and in light of the advances done in the Machine Learning area, some new proposals have appeared [4, 6]. These works first characterize SPARQL queries based some of their structural characteristics, and the learner component learns about the underlying properties of queries and data. For instance, these works learn about the query algebra features, graph pattern features or plan-level features for representing them as vectors. Next, they run several Machine Learning algorithms to predict the query performance.

In this work we propose the use of Dense Neural Networks to improve such SPARQL query processing times. We also present experimental results comparing our approach with other state of the art approaches, using both synthetic and real SPARQL queries. We show that our proposal improves the performance of other Nu-SVR based approaches by validating the results using using the Root Mean Squared Error metric. We also generated a new dataset with almost 19,000 training samples from real execution query logs. Results show that neural network models are good for the current problem obtaining more generalization using different datasets.

The rest of the paper is organized as follows. In Section II we introduce the reader to the concepts we use within the paper. In Section III, we discuss related work looking also at the work done in relational database systems. In Section IV, we describe the feature extraction techniques used to represent the queries. In Section V we describe the datasets, models and validation metrics used in the comparison, as well as the experiments performed. In Section VI we present the results

of our experiments ad finally, in Section VII we draw some conclusions and point out the directions of future work.

II. Background

In this Section we introduce some of the necessary background to understand the rest of the paper, including key concepts of SPARQL, RDF and Machine Learning.

A. RDF & SPARQL

RDF [11] is the graph-based data model at the heart of the Semantic Web. RDF terms can be IRIs (I), literals (L) or blank nodes (B). $(s, p, o) \in (I \cup B) \times I \times (I \cup L \cup B)$, is called an RDF triple, where s is called the subject, p the predicate and o the object of the triple. An RDF graph is a set of RDF triples. SPARQL is the standard query language for RDF, which have the usual set operations (AND, UNION, SELECT). Let V be a set of variables. A tuple $t \in P = (I \cup V \times I \cup V \times I \cup L \cup V)$ is called a triple pattern. Blank nodes in triple patterns can be considered as query variables for our purposes. A set of triple patterns is called a basic graph pattern P. We denote by $\mathrm{var}(tp)$ and $\mathrm{var}(P)$ the set of variables found in a triple pattern t and basic graph pattern P, respectively. We call a variable $?X \in \mathrm{var}(P)$ a join variable if it appears in two or more triple patterns of P, and a lonely variable otherwise. The semantics of SPARQL queries is defined in terms of mappings. A mapping μ is a partial function $\mu : V \rightarrow I \cup B \cup L$. The domain of μ, denoted $\mathrm{dom}(\mu)$, is the set of variables in which μ is defined. Given a triple pattern t, we denote by $\mu \in t$ or $\mu \in P$ the image of the triple pattern t under μ: the triple obtained by replacing the variables in t according to μ. We say that two mappings μ_1 and μ_2 are compatible, denoted $\mu_1 \sim \mu_2$ when for all $?X \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$ it is the case that $\mu_1(?X) = \mu_2(?X)$. Given sets of mappings Ω_1 and Ω_2, we then define their join as Ω_1 AND Ω_2, their union as Ω_1 UNION Ω_2 and the OPTIONAL as Ω_1 OPT Ω_2, as defined in [28] (the OPTIONAL operator is similar to a left outer join in relational databases). Finally, SPARQL also considers filtering out mappings using the FILTER operator and built-in conditions: P is a graph pattern and R is a SPARQL built-in condition ($<$, $>$, $=$), then the expression (P $FILTER$ R) is a graph pattern (a filter graph pattern).

B. Machine Learning algorithms

Tom Mitchel defined a machine learning algorithm as "A computer program is said to learn from experience **E** with respect to some class of tasks **T** and performance measure **P**, if its performance at tasks in **T**, as measured by **P** , improves with experience **E**".

Machine learning algorithms can be broadly categorized as unsupervised or supervised by what kind of experience(**E**) they are allowed to have during the learning process. Particularly, a supervised learning algorithm learns a function or *hypothesis* $f : \mathcal{X} \longrightarrow \mathcal{Y}$ from a set of examples $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_M, y_M)\}$. Our goal is to predict the output associated with a new input $\mathbf{x}$.

When $\mathcal{Y} = \mathbb{R}$ we have a regression task, i.e, we learn a hypothesis $f : \mathbb{R}^n \rightarrow \mathbb{R}$ [13]. Among the models commonly used in this task are: Linear Regression, a type of Support Vector Machines known as SVRs, and Neural Networks adjusted for regression problems. In our case, an input example consists of a set of features that represent a SPARQL query and its execution time is the target. To evaluate the abilities of a machine learning algorithm, we must design a quantitative measure of its performance. Usually this performance measure **P** or *loss function* is specific to the task **T** being carried out by the system[13]. For example, for regression settings the Mean Squared Error(MSE) is commonly used. Considering all pairs $(\mathbf{x}, y)$ from the train set S, it measures the average of the squared errors of each prediction $f(\mathbf{x}_i) = \hat{y}_i$ with respect to its original target y_i:

$$MSE_{(test)} = \frac{1}{m} \sum (y_i^{(test)} - \hat{y}^{(test)})^2 \tag{1}$$

The superscript $(test)$ on each term, emphasizes that the evaluation of the metrics must be reported on a dataset isolated from the data used in training.

III. Related Work

SPARQL query processing systems suffer when they receive thousands of queries in short periods of time [9], which is usually the case [8]. Also, SPARQL systems receive complex queries [8], that require efficient query executions (in time and space). Unfortunately, it is a hard task to efficiently execute such queries since they are very heterogeneous, and in practice impossible to generate efficient query execution plans for all of them.

The query optimizer is the database component in charge of generating efficient query execution plans. Traditionally these optimizers rely in two main strategies: heuristics using statistical metadata about the stored data and a dynamic programming strategy for selecting the best possible query plan. That strategy is followed by most SPARQL engines, since they rely on a relational database query optimizer (such as the widely used engine Virtuoso [12]). These systems collect statistics about the data like histograms however they have the disadvantage in the RDF data about the unpredictable data due to the differences in the data model [31]. Statistics are missing about all concepts in the database, and there is no clear schema. Another approach for efficiently accessing data is to index all of it and using heuristics for generating query plans (such as Jena TDB [1]). These techniques, generally do not require any statistical knowledge about the data, however these systems are not as efficient as the ones relying in data statistics (indexing depends on an efficient way of dealing with index buckets). We can conclude that there is a fair amount of space for improving query execution performance in RDF databases.

Recently several works have appeared that try to improve SPARQL query optimizers using Machine Learning techniques. One of the earliest is [16], in which the authors

propose the use of Support Vector Machine for regression (Nu-SVR) for predicting efficient query plans. The authors encode the SPARQL query characteristics (the SPARQL algebra) as vectors. The goal of the model is to predict which query plan will have the fastest execution. However, the authors do not take into account many details about SPARQL query shapes and the heterogeneity of the data stored, and thus their query representation is not complete, leaving much room for improvement. This is a key concept for the Machine Learning model, since without a proper characterization of the data and queries it is impossible to provide accurate predictions.

In the relational database systems community there is an intense work in this specific area. Currently, many research groups focus on using machine learning techniques to improve the efficiency of the query execution process. Recent works range from index prediction [32] to improve the query plan generation using Deep Reinforcement Learning [20], representing statistics for its use in deep reinforcement learning models [27], Deep learning models to predict metadata (statistics) for the query optimizer [18], etc. In the next lines we summarize some of the most important works in the area, so they can provide some insight to our purpose. However we can make a safe statement about these works: all focus on providing a good characterization of the data and queries to optimize.

The work in [18] uses deep learning to predict data correlations (join-crossing) within the data, tackling some weak points in techniques like basic per-table statistics. The proposed framework is called Multi-Set Convolutional Network (MSCN) is based on the Deep Sets work [33] which allows to express query characteristics using sets, which later can be used as input for the deep learning models. This is particularly useful since heuristics are based on statistics about sets of data. The authors also use Deep Sets to characterize the data in the database tables, enriching training data with information about these tables or bitmaps (annotate each table in a query with the positions of the qualifying samples represented as bitmaps).

Though the approach helps in finding good cardinality estimates (key input for the optimizer) it does not help directly in finding a good query plan (remain still thousands of possible query plans). To describe the SQL queries the authors use sets for tables, joins and predicates (boolean expressions within the query). Tables and joins between tables are represented as one-hot vectors while predicates of the type (*col, op, val*) are represented using a one-hot vector for *col* $+$, a one-hot vector for *op* $+$ *val* using a normalized value $\in [0, 1]$. Using this characterization the authors manage to obtain better cardinalities estimates than state of the art systems.

The work [21] generate optimized query plans by generating the execution cost of each subquery within a query. The authors define a subquery as a single join between two tables. The authors use a recursive model (NN_{ST}) which is based on the characteristics of each query, which are:

- Every state H_t represents a subquery composed by x_t and an action a_t where x_0 is information about each database properties (min/max values, num of distinct values, 1D histogram) whereas a_0 is a single relational operator($=\neq<>\leq\geq$).
- The next states are mapped as $x_{t+1} = H_t$ and a_{t+1} = another single relational operators like ($\bowtie$).
- Finally (NN_{ST}) calls $(NN_{Observed})$ which maps from hidden state to observed variables at time t in a supervised learning process using as exit the query cardinality.

.

In Neo [24], the authors approach the problem of optimizing queries from an end-to-end point of view: the system learns to make decisions about join order, physical operator, and index selection. Neo groups some of the most useful techniques for encoding queries as vectors and histograms (using some of the previous approaches) and proposes a new embedding from extracted automatically by using data from the underlying database. Neo also encode query plan characteristics as trees. The previous encoding is processed by a deep neural network called 'Value Network' which uses Tree Base Convolutional [25], and can predict the query or subqueries execution performance. Its performance compares to the Microsoft SQLServer or Oracle query optimizers.

As we have seen in the previous works, one of the most important phases in using machine learning techniques for predicting the performance of query plans is finding the right representation for the queries and for the database on which the queries are executed. All the works described [24, 21, 18, 27, 16] characterize queries in vectors, and based their predictions mostly on these characterizations. In the next Section we present our own characterization for SPARQL queries and RDF databases.

IV. SPARQL Query Characteristics

In order to feed a machine learning algorithm with SPARQL queries, we need to represent each query as an input vector with the query characteristics that best represent them. We implement two different vector representations of the SPARQL queries characteristics, proposed by [16].

A. SPARQL Algebra Features

Our first approach use the SPARQL operators frequencies in the queries as characteristics. More in detail we select as characteristics the amount of BGPs, joins, OPTIONAL, UNION and FILTER operators. Other operators are less used [8] and may introduce unnecessary noise to the model. We also include as query characteristics the number of triple patterns in the queries. Figure 1 shows a sample of our newly vector approach.

Instead of only including join occurrences between BGPs we also include joins between triple patterns. This allows to include more information into the model, since most joins happen between the triple patterns within the queries. It is worth noticing that when using the FILTER operator the SPARQL performs a selection of the data. To capture such data selection we include as features the filtering byURI, numeral and literal. That filtering may also happen in object of subject position in triple patterns.

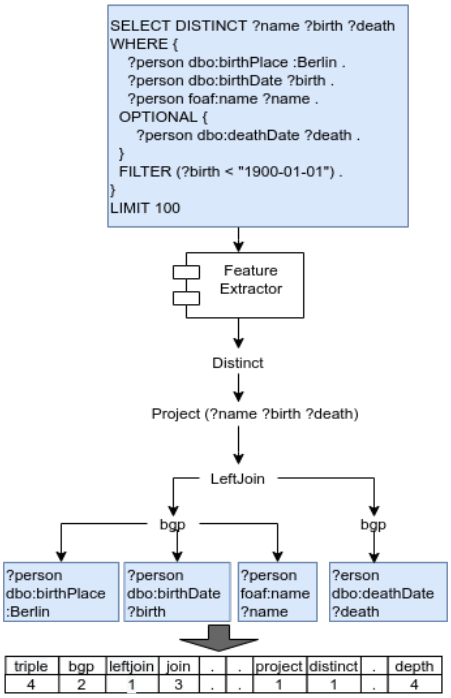

Figure 1. Extracting SPARQL algebra features from a SPARQL query(Image available in [16]))

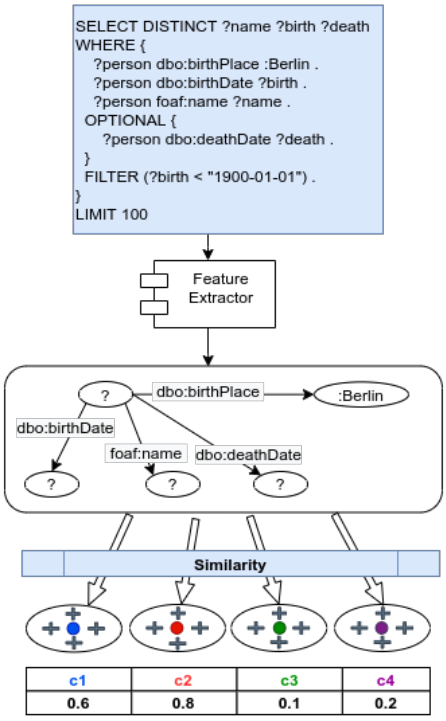

Figure 2. Example of extracting graph pattern features.

B. Graph Pattern Features

The second approach benefits from the ability of modeling SPARQL queries as query graphs. This approach uses similarity patterns between queries represented as graphs using a vector of K_{gp} dimensions, where each dimension value is the structural similarity between that query pattern and any of the other K_{gp} representative patterns in the dataset.

Figure 2 shows an example of SPARQL query feature extraction, we describe the process in the following lines:

1) The first step is to build a graph that is representative for a SPARQL query (a triple pattern represents two nodes and an edge between these nodes). Next, each edge and vertex is replaced by a symbol.
2) The second step is to make clusters from the queries using the k-medoids [17] algorithm. K-medioids chooses the centroids (K_{gp} centroids) using a distance function like Edit-Distance. The 'Clustered training queries' shows the query groups identified. Each circle represents a query cluster in which each centroid is represented using the blue color.
3) Finally, to obtain the vector representation of K_{gp} (the query pattern), we calculate the structural similitude between a query graph pi and the k-ith center of the cluter's query graph $C(k)$.

$$sim(p_i, C(k)) = \frac{1}{1 + d(p_i, C(k))} \quad (2)$$

The term $d(p_i, C(k))$is the graph's edit distance between the query graphs p_i y $C(k)$, obtaining a similitude score between 0 and 1 (being 0 the most different and 1 the most equal).

The results from [16] propose the combination of both approaches la (SPARQL Algebra Features + Graph Pattern Features) to get a better performance. We use that combination to generate our dataset and train the proposed model in the next Section.

V. Data and Models

In this section we present the datasets used to evaluate our approaches. We describe the quality of the datasets used to train machine learning models and the motivation to generate a new data set for the evaluation.

A. Data

The authors in [15] provided the dataset used in [16] which was processed as described in Section IV. For purposes of reproducibility, this dataset is standarized and divided in train, validation and test sets of 1260, 420 and 420 query vectors respectively.

However, looking into the train set we can observe that from the 1260 examples, there are only 60 different and unique vectors. This implies that the other train examples are exact copies of some of these 60 unique vectors. This issue motivate us to generate a dataset with a greater diversity of examples. For this, we select 20,000 queries from LSQ [29]. Where a 33% are queries with an execution time of at least 100ms, another 33% have an execution time between 100ms y 1000ms and the other need more than 1000 ms to be executed. In this way, we balance the execution time of the queries that we include in the dataset. The feature vectors are the results from applying the techniques proposed by [16]. We finally obtained

a total of 18948 vectors and we divided the dataset in the following manner:

- Vectors in the **train** set: 9947.
- Vectors in the **validation** set: 4264.
- Vectors in the **test** set: 4737.

We check the unique values present in the new generated dataset, finding that of the 9947 of the training set there are only 515 different vectors. This implies, just like in [15], that the rest of the vectors are exact copies of some of these groups. The rates of unique values with respect to the total of training examples remain similar: 0.047 (in [15]) and 0.051 in our dataset.

After noticing the above, we decided to explore the quality of the data a little more. Based on the fact that there are few unique values, it is necessary to verify that the targets associated with these identical vectors are at least similar. Accordingly, for each dataset, we group the identical input vectors (those with exactly the same feature values). Next, we analyze how the associated targets with them vary.

Figures 3 and 4 show boxplots of the execution times for 10 randomly selected groups for data from [15] and our generated data respectively. Indeed, it can be seen that exists in both datasets a significant number of identical vectors that are far from the average of their group values. These points are outliers that affect the performance of any model trained over this data.

B. Models

In [16] the authors experimented with two regression models: one using k-Nearest Neighbors (k-NN) [5] and another using a Support Vector Machine (SVM) with a nu-SVR nucleus for regression as in [10]. The Nu-SVR, with parameters $C = 300$ and $\mu = 0.3$ [16], obtained a better result with a Root Mean Squared Error(MSE) of 262.1869, with a Determination Coefficient (R^2) of 0.98526 in the train set. Thus, it was selected as baseline.

We hypothesize that using dense Neural Networks we can obtain models with a better capacity for generalizing the previous results. The parameters we used for building our neural network are the following:

- The number of neurons in the input layer is set to 34, one per dimension for each vector: 1 dimension for each SPARQL query feature plus 9 for each the SPARQL algebra feature encoded.
- We tune the network architecture, i.e., the number of hidden layers, the number of units of each hidden layer and the activation functions.
- In the output layer we use only a single neuron since we are targeting a regression problem. We will test with linear and sigmoid activation functions to select the best performance one for the $target$.

We also explore the a data preprocessing stage using and autoencoder with the following characteristics:

- Use a dense neural network to model the auto-encoder.
- Use 1 hidden layer to encode the information with 20 units. Followed by a reconstruction layer of the input with the same number of neurons as the input.
- Use Mean Squared Error as the loss function.
- As an optimizer we test the best performance among Adam, Nadam, SGD.

After this preprocessing we train the previous model and evaluate its performance comparing with the baseline and the model without preprocessing.

C. Validation

For validation we compare the results from the baseline using the Root Mean Squared Error (RMSE) metric:

$$RMSE = \sqrt{\frac{\sum(y_i - \hat{y})^2}{N}}. \tag{3}$$

As its name indicates, it is the square root of the average of the magnitude of the quadratic error obtained.

In [16] the metrics of R^2 are also presented. However, we decided not to include them in the comparison with our proposal. The exclusion is based on the fact that R^2 is not suitable enough for assessing nonlinear models [3].

We use the train and validation sets to train and select the hyperparameters of the compared models (proposal and baseline). Then we evaluate and report the results using the test set which has not been observed during training and model selection. In this way we guarantee that the performance obtained is representative of the rest of the untrained queries.

D. Software and Hardware

We use Apache Jena 2.11.0 libraries to extract the characteristics of LSQ Sparql queries. Dataset manipulation, model training, and evaluation were developed in the Jupyter notebooks available at [7, 2]. To train the models, we use an AMD Opteron Server with 24 cores 2.8 GHz, 64 GB system RAM, a GeForce RTX 2080 graphics card with 2GB, CUDA Version: 10.1. Linux Operating System version 5.0.0-37.

VI. Experimental Results

In this Section we present the results for our approach. The first comparison corresponds to the results from comparing our approach and with the results using the data in [16]. next, we present the results using our generated dataset from the queries in [29].

A. Original dataset results

To reproduce the results presented in [16], the Nu-SVR model was tuned using the original parameters ($C = 300$ and $\nu = 0.3$).

For assessing our proposal we used a grid search to tune the hyper-parameters and select the best ANN model using the following combinations:

- We train the models with 3 hidden layers, with between 80 and 120 neurons per layer.
- As optimization function we used a Stochastic gradient descent (SGD) with a learning_rate of 0.001.

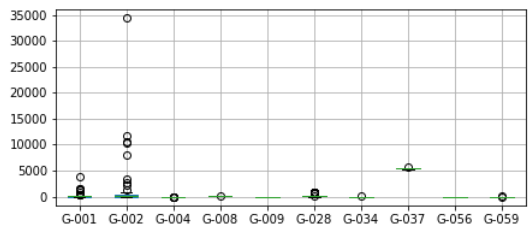

Figure 3. Boxplot with data from [15]. In the x axis we show the train examples grouped and the y axis show the targets distribution by group.

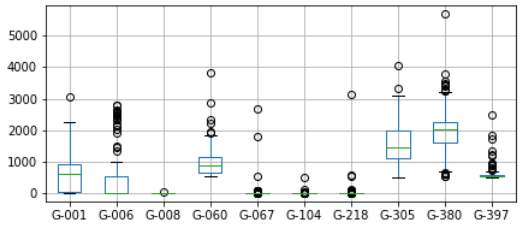

Figure 4. Boxplot with generated data from queries in LSQ [29]. In the x axis we show the train examples grouped and the y axis show the targets distribution by group.

Metrics	Train set	Validation set	Test set
R^2	0.735595	0.940999	0.98513
RMSE	1279.93092	534.5756	263.27768

Table I
RESULTS OBTAINED AFTER REPLICATING THE BASELINE NU-SVR MODEL WITH $C = 300$ AND $nu = 0.3$

.

L1	L2	L3	MSE_TRAIN	MES_VAL
80	60	60	1276.2749	526.3932
80	70	70	1276.8321	526.3820
85	**60**	**60**	**1274.4424**	**522.6416**

Table II
L1, L2, L3 ARE THE NUMBER OF UNITS USED IN HIDDEN LAYERS 1, 2 AND 3 RESPECTIVELY. EVALUATION ON ORIGINAL DATASET.

C	Nu	RMSE_TRAIN	RMSE_VAL	TRAIN_TIME
200	0.45	1074.3239	1004.0438	590.460
220	0.45	1074.3221	1004.0431	705.098
260	0.40	1074.3224	**1004.0429**	667.859

Table III
BEST 3 RESULTS FOR THE NU-SVR IN GENERATED DATASET. THE TIME COLUMN CORRESPONDS TO THE TRAINING TIME IN SECONDS.

L1	L2	L3	RMSE TRAIN	RMSE VAL
260	280	240	634.7420	573.8813
300	380	260	635.1847	572.9962
340	380	340	598.2774	**543.0992**

Table IV
GRID SEARCH TO SELECT THE NETWORK ARCHITECTURE. THE ROWS IN THE TABLE ARE ORDERED IN DESCENDING ORDER BY THE COLUMNS RMSE_VAR.

- As loss function we used MSE.
- We trained a 200 epoch as maximum, configured with an EarlyStopping having a patience of 20 to be able to stop the training in the case of having 20 wrong consecutive results.
- We standardized the data using StandardScaler from the Sklearn library.
- We used a logarithmic scale for the targets, and next they were rescaled for computing the RMSE.

Table II shows the performance of the grid search to select the hyperparameters of the model.

The best RMSE in the test set is **251.5771**. These results are slightly better than the ones using Nu-SVR.

We see that in both models the RMSE from the train set are worse than the validation set. The same happens with the test set with respect the validation set. This shows that the data quality is bad since what should happen is the contrary.

B. Results in the generated dataset

Table III depicts the performance between baseline with Nu-SVR for our generated dataset. We run 40 iterations using random values for the hyperparameters C y nu. The values for C were generated in the range $[100 : 300]$ and for nu in the $[0.1 : 0.5]$.

We proceeded similarly for training another multi layer neural network using gridsearch and the following parameters:

- The amount of neurons per layer are between 220 and 400.
- The optimization function we used Adam with a learning_rate of 0.00025.
- As loss function we used Mean Square Error.
- As regularization technique we used a Dropout layer between each hidden layer fix a fixed ratio of 0.25.
- We set 450 epochs with an early stopping with a patience setting of 20.
- The data were standardized with StandardScaler from the Sklearn library.
- As before, we use logarithmic scale for targets, then they are rescaled for computing the RMSE.

Table IV shows the best results in the validation set. We obtained the lowest RMSE value **543.1**.

C. Results using an autoencoder for preprocessing data

We evaluate the use of an autoencoder of 30 units in the hidden layer for preprocessing data. We obtained the best results in auto-encoder training using the following configuration:

- Using Adam with Nesterov momentum as optimizer with a learning rate of 0.0001.
- 300 epochs were trained with an early stopping with patience of 10.

Figures 6 and 7 show the trained architecture and the loss performance with respect to the amount of epochs in the auto-encoder training.

Next, we used the encoder layers as input to train the best model presented in the previous Section. Figure 8 shows the

Modelo	TRAIN	VAL	TEST	TIME
Nu-SVR	1074.3224 (0.000047)	1004.0429 (0.000021)	946.3783 (0.000032)	667.859
ANN	625.6216 (524.6459)	560.0981 (299.9600)	**461.2723** (371.8330)	149.429
AEC+ANN	634.56 (949.1886)	568.7562 (445.1719)	479.3260 (591.0072)	346.278

Table V
RMSE RESULTS AND TRAINING TIME FOR: TRAINING, VALIDATION AND TESTING SETS FOR EACH MODEL.EVALUATION ON THE GENERATED DATASET. THE VALUES IN PARENTHESES CORRESPOND TO THE VARIANCES BETWEEN THE 10 RUNS FOR EACH MODEL AND SET.

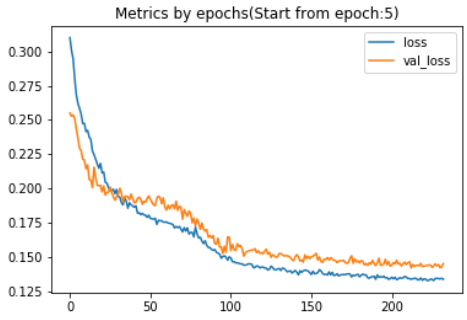

Figure 5. RMSE values for the training and validation sets and better selection of the hyper-parameters.

architecture we obtained after adding the encoder layer as the first layer of the architecture.

Finally, **Table V** shows the results for the 3 models tested. We performed 10 runs of our proposed models and averaged the results to report them. We also report variances for each model in each set (between parentheses). Our results show that the proposed models are competitive using the original dataset and better than the competition using the newly generated dataset. Furthermore, we can train our model faster than using the Nu-SVR. We also think that there is room for improvement in our model. As we will discuss in the next Section, beyond the model used the quality of the characteristics extracted is the main goal of future work.

VII. CONCLUSIONS

In this work we presented an approach for improving query execution times of SPARQL queries by using Deep Learning techniques. We focused first on finding a good characterization of SPARQL queries and RDF data for next implementing a Feed Forward model that allowed us to predict the performance of each new query received by the system. We also generated a new and larger dataset that allowed us to show that Feed Forward models are actually better than the models by [16] with Nu-SVR.

Analyzing the test suite, we show that neural models gain better generalization capabilities with new data. In this case, the best performance is presented by the neural model without pre-training.

In the new dataset we also see that validation and test sets obtain lower RMSE values than the training set, which is contradictory (should be the opposite). However, the differences are not as significant as in the first dataset. We believe that our generated data is of better utility to tackle the current problem.

Beyond the model used, we identified several problems that correspond to the characteristics identified:

- The extracted features retrieve only information from the query structures. They do not include information such as the predicates used, joins in predicates, the order of execution between the triples in the used Query Execution Plan.
- We evaluate the quality of the data and identify the presence of outliers. Examples with same vector containing quite different execution times affect the model learning process. The same behavior was observed in the generated dataset. This validates our idea that the problem is that the characteristics used are not enough for solving the problem.

Beyond the approach presented, query optimization largely depends on cardinality estimates. To be able to select a good query plan it is necessary to obtain good intermediate result estimates for the joins within the query. We aim at improving our approach by following and adapting some of the ideas using Deep Sets [20] from [18]. As the authors say, the biggest challenge in cardinality estimation are join-crossing correlations. However, the cardinality estimations need to be used by a query optimizer, which is the component that finally proposes a query plan to be executed. Thus, it is necessary to integrate these estimations, which we will do in the next steps.

In our future work we aim at combining a cardinality estimation approach using our approach combined with some of the Neo's techniques, however this is far from trivial since there are no tables is SPARQL, nor any cardinality estimation. As an example, for 18 thousand queries from LSQ we identified 250 different predicates, implying Join Graph features with high dimensional vectors. Due to this high dimension it is impossible to encode all predicates using one hot vectors. Similarly, to generate new embeddings we may obtain $100 * n$ with n = number of predicates.

REFERENCES

[1] Apache Jena, https://jena.apache.org/
[2] Dacasals/sparql-query-exec-prediction-model, https://github.com/dacasals/sparql-query-exec-prediction-model
[3] An evaluation of R 2 as an inadequate measure for nonlinear models in pharmacological and biochemical research: A Monte Carlo approach — SpringerLink, https://link.springer.com/article/10.1186/1471-2210-10-6
[4] A Machine Learning Approach to SPARQL Query Performance Prediction - IEEE Conference Publication, https://ieeexplore.ieee.org/document/6927552

Layer (type)	Output Shape	Param #
input_31 (InputLayer)	(None, 44)	0
dense_385 (Dense)	(None, 30)	1350
dense_386 (Dense)	(None, 44)	1364

Total params: 2,714
Trainable params: 2,714
Non-trainable params: 0

Figure 6. Architecture of the best trained autoencoder.

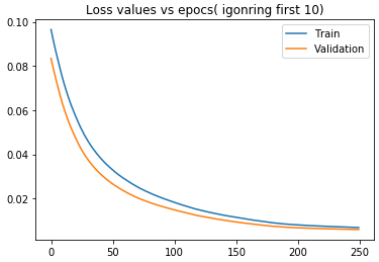

Figure 7. MSE values for the training and validation sets of best auto-encoder selected.

Layer (type)	Output Shape	Param #
dense_391 (Dense)	(None, 30)	1350
dense_393 (Dense)	(None, 340)	10540
dropout_179 (Dropout)	(None, 340)	0
dense_394 (Dense)	(None, 380)	129580
dropout_180 (Dropout)	(None, 380)	0
dense_395 (Dense)	(None, 340)	129540
dense_396 (Dense)	(None, 1)	341

Total params: 271,351
Trainable params: 271,351
Non-trainable params: 0

Figure 8. Feedforward model architecture with the first two layers of the auto-encoder.

[5] Aha, D.W., Kibler, D., Albert, M.K.: Instance-based learning algorithms **6**(1), 37–66

[6] Akdere, M., Çetintemel, U., Riondato, M., Upfal, E., Zdonik, S.B.: Learning-based Query Performance Modeling and Prediction. In: 2012 IEEE 28th International Conference on Data Engineering. pp. 390–401. IEEE. https://doi.org/10.1109/ICDE.2012.64

[7] Amat, D.A.C.: Dacasals/sparql-query2vec, https://github.com/dacasals/sparql-query2vec

[8] Bonifati, A., Martens, W., Timm, T.: An analytical study of large sparql query logs. The VLDB Journal pp. 1–25 (2019)

[9] Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: Sparql web-querying infrastructure: Ready for action? In: International Semantic Web Conference. pp. 277–293. Springer (2013)

[10] Chang, C.C., Lin, C.J.: LIBSVM: A library for support vector machines **2**(3), 27

[11] Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation (Feb 2014)

[12] Erling, O., Mikhailov, I.: Rdf support in the virtuoso dbms. In: Networked Knowledge-Networked Media, pp. 7–24. Springer (2009)

[13] Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), http://www.deeplearningbook.org

[14] Harris, S., Lamb, N., Shadbolt, N., et al.: 4store: The design and implementation of a clustered rdf store. In: 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009). pp. 94–109 (2009)

[15] Hasan, R.: Learning Sparql Query Performance from DBPedia Query Logs. Part of My PhD: Rhasan/Query-Performance https://github.com/rhasan/query-performance

[16] Hasan, R., Gandon, F.: A machine learning approach to SPARQL query performance prediction. In: 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT). vol. 1, pp. 266–273. IEEE

[17] Kaufman, L., Rousseeuw, P.J.: Finding Groups in Data: An Introduction to Cluster Analysis, vol. 344. John Wiley & Sons

[18] Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P., Kemper, A.: Learned Cardinalities: Estimating Correlated Joins with Deep Learning http://arxiv.org/abs/1809.00677

[19] Kraska, T., Alizadeh, M., Beutel, A., Chi, E.H., Ding, J., Kristo, A., Leclerc, G., Madden, S., Mao, H., Nathan, V.: SageDB: A Learned Database System p. 10

[20] Krishnan, S., Yang, Z., Goldberg, K., Hellerstein, J., Stoica, I.: Learning to Optimize Join Queries With Deep Reinforcement Learning (2019), http://arxiv.org/abs/1808.03196

[21] Krishnan, S., Yang, Z., Goldberg, K., Hellerstein, J., Stoica, I.: Learning to Optimize Join Queries With Deep Reinforcement Learning (2019), http://arxiv.org/abs/1808.03196

[22] Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., Neumann, T.: How good are query optimizers, really? Proceedings of the VLDB Endowment **9**(3), 204–215 (2015)

[23] Malyshev, S., Krötzsch, M., González, L., Gonsior, J., Bielefeldt, A.: Getting the most out of wikidata: semantic technology usage in wikipedia's knowledge graph. In: International Semantic Web Conference. pp. 376–394. Springer (2018)

[24] Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., Tatbul, N.: Neo: A learned query optimizer **12**(11), 1705–1718 (2019). https://doi.org/10.14778/3342263.3342644

[25] Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z.: Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the

Thirtieth AAAI Conference on Artificial Intelligence. pp. 1287–1293. AAAI'16, AAAI Press

[26] Neumann, T., Weikum, G.: Rdf-3x: a risc-style engine for rdf. Proceedings of the VLDB Endowment **1**(1), 647–659 (2008)

[27] Ortiz, J., Balazinska, M., Gehrke, J., Keerthi, S.S.: Learning State Representations for Query Optimization with Deep Reinforcement Learning http://arxiv.org/abs/1803.08604

[28] Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. ACM Transactions on Database Systems (TODS) **34**(3), 1–45 (2009)

[29] Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.C.N.: LSQ: The linked SPARQL queries dataset. In: Arenas, M., Corcho, O., Simperl, E., Strohmaier, M., d' Aquin, M., Srinivas, K., Groth, P., Dumontier, M., Heflin, J., Thirunarayan, K., Staab, S. (eds.) The Semantic Web - ISWC 2015. pp. 261–269. Springer International Publishing

[30] Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of the 1979 ACM SIGMOD international conference on Management of data. pp. 23–34 (1979)

[31] Tsialiamanis, P., Sidirourgos, L., Fundulaki, I., Christophides, V., Boncz, P.: Heuristics-based query optimisation for SPARQL. In: Proceedings of the 15th International Conference on Extending Database Technology. pp. 324–335. ACM

[32] Wang, W., Zhang, M., Chen, G., Jagadish, H.V., Ooi, B.C., Tan, K.L.: Database meets deep learning: Challenges and opportunities. SIGMOD Rec. **45**(2), 17–22 (Sep 2016). https://doi.org/10.1145/3003665.3003669, https://doi.org/10.1145/3003665.3003669

[33] Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R.R., Smola, A.J.: Deep sets. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems 30, pp. 3391–3401. Curran Associates, Inc., http://papers.nips.cc/paper/6931-deep-sets.pdf