Abstract

Today's data analytics systems (DASs) are commonly-used to run ML programs and SQL-like programs, which we call workloads. Tuning such DASs to optimize a user objective (e.g., latency) is non-trivial due to DAS's complex system configuration parameters. While previous work mainly uses a high-cost error-and-trial approach to find a good system configuration, this project focuses on developing deep learning models to predict the user objective, and hence makes it possible to recommend decent DAS configurations for a workload afterwards. Ideally, we hope to use a classical neural network regressor to predict the value of a user objective with the DAS configuration and workload information as its inputs. However, while the configuration is easy to be represented as the vectorized configuration parameters, the workload is difficult to be explicitly expressed. Hence, we further apply an autoencoder model to encode the workload. The input and output of our autoencoder is the traces we collected when running workloads under configurations, which we call observations. As both the configuration and workload contribute to the bottleneck layer of an autoencoder model, we split the bottleneck layer into two parts, and extract the workload encoding from one. The autoencoder loss function embeds the Euclidean distance between one part of bottleneck layer and the real configuration parameters to squeeze the workload representation into the other part.

We collect training data from the traces when running different SQL-like and ML workloads under variant configurations. We use Mean Absolute Percentage Error (MAPE) to capture the model accuracy. Currently, the lowest MAPE of SQL is 35.94%, and we are still tuning the models to push it lower.

1. Introduction

Today’s data analytics systems (DASs) are commonly-used to run dataflow programs like ML programs and SQL-like programs, which we call workloads. However, tuning such DASs like Apache Spark [13] and Apache Flink [3] are non-trivial because of their complex system configuration parameters. Taking Apache Spark for example, the configuration parameters include parallelism (for reduce-style transformation), RDD compression (Boolean), memory per executor, memory fraction (of heap space), Batch interval (the size of each mini-batch) and Block interval (the size of data handled by each map process) in the streaming setting, to name a few.

One of the biggest challenges in tuning such a system is that many configuration parameters are not independent (e.g., changing one can impact others), not universal (e.g., what works for one workload may be a sub-optimal choice for another) and with infinite choices (e.g., some parameters are numeric). When tuning a DAS, expert engineers usually look into a workload first and start to use an error-and-trial approach to find a good configuration. However, they can not handle every scenario because workload and configuration are sometimes beyond the understanding of a human being. Hence, it is very hard, even for an expert engineer, to provide the DAS with the right parameter choices, as also shown in a recent study [9]. In the real world, companies still needs to hire engineers to tune their systems, which is a big cost both in money and time. Therefore, tuning DASs automatically is super valuable and important.

This project mainly focuses on how to develop deep learning models to map information from the workload and the configuration to the value of an user objective (e.g., latency or throughput). Based on the models, our ultimate goal is to recommend a good configuration to a workload automatically to achieve the best user objective. Basically, the idea is to (1) regard the given job as a black box and develop deep learning models, as accurately as possible, to predict the user objective; (2) based on the models, evaluate the user objective under different configurations, and recommend a good configuration accordingly. To clarify, in this project, we ignore the effect from hardware and software deployment by running every workload under the same environment (e.g., a particular EC2 instance), and we also regard the configuration recommending part as our future work.

Ideally, we want a classical neural network regressor
(NNR) to predict the value of a user objective by regrading two independent features, the workload (of dataflow programs) and the configuration, as the input, and the user objective, as the output. However, while the configuration is easy to be represented as the a series of parameters, the workload is more obscure and more difficult to be directly expressed.

To represent workload, our project first collects the runtime traces, collectively called observations, during executing a dataflow program. The traces collected in an observation include three types: (i) application-level metrics, (ii) OS-level metrics and (iii) the value of the user objective.

Next, we take advantage of the feature representation property of the deep learning to extract the workload encoding from the runtime traces by using an autoencoder (AE) model. By applying AE model, we will get a bottleneck layer in the middle of the network, which can be represented as the combination of workload and configuration. We further split the bottleneck layer into two parts. The first part is considered as the workload encoding, while the second part is supposed to be the same as the related configuration parameters. To squeeze the workload representation into the first part of the bottleneck layer, the loss function of our AE model embeds the Euclidean distance between the second part of the bottleneck layer and the real configuration parameters. After having the workload encoding, we can go back to the NNR model to predict the user objective.

The overall tuning process contains two parts. The first offline part is like the training phase. The project uses collected traces as the training data set to train (1) an autoencoder model to encode the workload characteristics and (2) a neural network regressor to predict the user objective by using the workload encoding and the configuration. These two deep learning models are trained periodically and will not affect cost of the online part.

The other online part is like the testing phase and does not attend the training progress. When a new workload (dataflow program) comes in, we first run it for a small period of time (10-20 minutes) to get some traces. After traces are collected, the project (1) uses them as test data points of the trained AE model to get the workload encoding; (2) uses the workload encoding and the configurations as the test data points to get the user objective evaluation.

In our experiment, we collect more than 4,800 traces as the whole data set by running ML workloads and SQL-like workloads under variant configurations. We use Mean Absolute Percentage Error (MAPE) to capture the accuracy of our two models. Currently, the lowest MAPE for SQL-like and ML workloads are 35.94% and 76.49% respectively, and we are still tuning the models to make it lower.

2. Related Work

Scalable data analytics systems have gained widespread adoption in industry, including Hadoop, Spark [13], Naiad [7], etc. for general purpose analytics, and Hive [11], Scope [16], etc. for SQL queries. There are also a number of scalable stream analytics systems including Flink [3], Spark Streaming [14], Google Dataflow [1], Twitter Heron [6], and SAP HANA [4]. Most of these systems lack an optimizer that takes user objective and automatically chooses a system configuration to best meet the user objective. The approach proposed in our project has the potential to benefit all of these systems for automatic optimization.

The closet work to ours, Ottertune [12][15], optimizes configuration knobs of databases queries using a nearest neighbor” approach based on machine learning. By using the nearest neighbor result, Ottertune uses Gaussian Process (GP) regression to recommend configurations. While Ottertune still needs DBAs to select or measure the estimated results suggested from Ottertune and make a final decision, our project can directly provide the configuration to achieve the best user objective. Due to the lack of time, the comparison with Ottertune will be done as part of our future work.

3. Learning Model and Approach

Accurate models for a user objective are vital important. Based on the models, we can have an overall estimate for the chosen user objective, and ultimately recommend a good system configuration for a workload. In this section, we will first introduce the tuning process in our project. Then we define the problem in a general case, and finally introduce our approaches to predict a user objective.

3.1. Preliminary

When tuning the system for the best user objective, we first regard a running workload as a blackbox without internal information due to the frequent use of machine learning and user-define functions in today’s data analytics. In addition, to adapt to complex runtime behaviors, we propose to replace static modeling with in-situation modeling, which we defined as learning a model for a user objective in the same computing environment as the workload is executed.

To characterize our workload, we collect traces, collectively called observations, during workload execution. The traces we collect has following three types: (i) measurement of the user objective, e.g., latency or throughput; (ii) application-level metrics, e.g., from Spark in our prototype, including records read and written, bytes read and written, bytes spilled to disk, fetch wait time, etc.; (iii) OS-level metrics such as CPU, memory, IO, network usage collected using the Nmon command.
Tuning Data Analytics Systems Through Deep Learning
Chenghao Lyu
chenghao@cs.umass.edu

3.2. Tuning Process Overview

The design of our overall tuning process is comprised by two paths as shown in Figure 1. The red online path recommends a configuration based on our trained model. When a new workload comes, the trace collector will be running for a period of time (e.g., 10-20 minutes). Then in-situ modeling kicks in. It takes as input the new traces together with a model for encoding workload, which is trained previously from all past workloads. After getting the workload encoding, we can further search and recommend a good system configuration based on the user objective predicting model trained previously as well.

The brown offline path periodically retrains the models by taking all observational data from the past workloads, as well as the additional observational data from its internal benchmark to enrich training with a large set of different configurations.

Figure 1. Tuning Process Overview

To clarify, in this project, we mainly focus on the accuracy of our global model to predict the user objective. How to search and recommend a good configuration is part of our future work.

3.3. Problem Definition

When a user submits an ML or SQL-like program, she specifies a particular objective, and an initial configuration for DAS. During execution, workload \( j \) produces a trace, \( T_j^i(\theta_j, O_j) \), that corresponds to the current \((i^{th})\) configuration, \( \theta_j \), and includes the observation vector, \( O_j \). The observation vector has the application-level metrics, OS-level metrics and user objective value as mentioned in previous subsection. Without loss of generality, our work chooses latency as the user objective.

The goal of modeling is to build a predictive model for the user objective latency using collect traces set \( T \), and later use the model to predict the performance of latency for a new workload \( j' \) with a particular configuration \( \theta_j \).

Learning such models is challenging because a model depends not only on the configuration but also on the workload characteristics of each job. Hence, the modeling task needs to extract the workload encoding, \( W_j \), from the traces for job \( j \). Here, we define the workload encoding as follows.

**Definition 3.1** The Workload Encoding \( W_j \) is a compact real-valued vector that encodes the characteristics of a specific dataflow program (also called workload) on a given dataset.

Given the notion of workload encoding, we next define the learning problem. The model can be formulated as a deterministic function \( \Psi \) s.t.

\[
\Psi(\theta_j, W_j) = t_j
\]

(1)

The goal of our model is to choose the function that minimizes the distance between the predicted latency and the real latency for all (workload, configuration) combinations, called training instances, in the training data:

\[
\Psi^* = \arg \min_{\Psi} \text{avg}(\text{distance}(\Psi(\theta_j, W_j), t_j))
\]

(2)

where the average is taken among all the training instances, and configuration \( \theta_j \) can be under some constraints.

3.4. Approaches

Figure 2 shows how we predict the user objective based on the collected observations and configurations information. Ideally, if the workload and configuration can both be expressed, we can apply a classical neural network regressor (NNR) to model the user objective. To represent the workload, we use an autoencoder (AE) model to extract the workload encoding from its observation. Then we apply the NNR model to predict the user objective afterwards.

3.4.1 Autoencoder Model

An autoencoder model is a neural network architecture for non-linear dimensionality reduction to learn a compact representation of data. In our setting, the autoencoder transforms the observation \( O_j \) to a compact encoding \( E_j \), the bottleneck of the architecture. This encoding is learned by trying to reconstruct an approximation \( \hat{O}_j \) of the input \( O_j \), which naturally meets the reconstruction property.

Specifically, \( E_j \) is denoted as the compact encoding by the observation \( O_j \) when running workload encoding \( W_j \) under the \( j^{th} \) configuration. As the observation can be determined by the workload and configuration, the compact encoding \( E_j \) also expresses the combination of these two features.

Here, we divide compact encoding into two parts to present the workload and configuration respectively. Naturally, give a workload \( j \), the \( W_j \) should be fixed, even under variant configurations. To represent this property, we decomposes \( E_j \) into two parts. The first is a variant part \( E_j^Q \)
that approximates the job configuration $\theta^i_j$, with the expectation $|\tilde{E}^i_j| = |\theta^i_j|$. The other part aims to approximate the workload encoding as an invariant $\tilde{E}^i_j$, under different configurations. While the size of $\tilde{E}^i_j$ is fixed, the size of $\tilde{E}^i_j$ can be determined through hyper-parameter tuning.

To achieve the above goal during backpropagation, we introduce a second term to the loss function to guarantee the property of the workload encoding:

$$\text{Loss} = \frac{1}{m} \sum_{j \in J_{\text{train}}} \sum_{i \in I_j} (\|O^i_j - \tilde{O}^i_j\|^2 + \gamma \|\theta^i_j - \tilde{E}^i_j\|^2) \quad (3)$$

where $\gamma$ denotes a hyper parameter that controls how much the approximation of the job configuration contributes to the loss function. By minimizing the loss, backpropagation may be able to minimize both the data reconstruction error and the approximation error of the job configuration.

As for the workload encoding, $W_j$ can be denoted as the average of the $\tilde{E}^i_j$ from the running results under variant configurations:

$$W_j = \frac{1}{|I_j|} \sum_{i \in I_j} \tilde{E}^i_j \quad (4)$$

### 3.4.2 Neural Network Regressor

After encoding the workload, we can fit a regression function on both the job configuration $\theta^i_j$ and the extracted workload encoding $W_j$. Then we use a classical neural network regressor to do the evaluation for the user objective, as shown in the right side of the Figure 3.

### 3.4.3 Insights in Workload Encoding

Although we only have one metric as the models’ output, we regard the workload encoding as another implicit target.

Actually, we need workload encoding to have more future usage beyond the scope of this project. For example, when the user objective changes, we do not need to train the whole model from scratch. Instead, we can apply the existed workload encoding directly to another NNR model or fine-tune the autoencoder with a new NNR model to predict the new user objective.

In addition, we can mapping the relationship between different workloads to the workload encoding. Based on the relation, we can further borrow useful “knowledge” from close workload in our training set.

---

Figure 2. Overall Approach

Figure 3. Learning Models
The exploration on the workload encoding is also part of our future work.

4. Experiment

In this section, we evaluate our model and show the current results.

4.1. Setup

Since our work is pretty new, we design the benchmark by ourselves. In our experiment, we use Spark Streaming as our runtime system and latency as the user objective.

4.1.1 Runtime Environment

We chose Spark streaming as the runtime system to run workloads and collect traces. Spark Streaming is deployed in our 10-node cluster each with an Intel Xeon X3360 Processor (4 cores) and 8GB RAM.

For modeling, we implemented the neural network approaches using PyTorch libraries from Python 3.6. For prediction, we used the neural network regressor as the default option. We rent a Google Cloud server for modeling and training, which has 8 vCPUs processors, 30 GB of RAM and 1 Nvidia Tesla P100 GPU.

4.1.2 Dataset and Workload

We generated two types of workloads: SQL-like and Machine Learning (ML). The SQL-like workloads were generated from 5 parameterized templates for click stream analysis, which employ different group-bys, joins, global or windowed aggregates, and UDFs. We generated 52 workloads from these templates. We collected traces for 6 offline workloads (with an average of 488 job configurations), used exclusively for training, and 46 regular workloads (with an average of 32 configurations). The ML workloads use a template that applies binary classification using Stochastic Gradient Descent. We created 12 ML workloads by varying 3 parameters on regularization, and collected 32 configurations per workload as well.

4.1.3 Evaluation Metric

We use the Mean Absolute Percentage Error (MAPE) metric to capture model accuracy:

\[
\text{MAPE} = \frac{100}{m} \sum_{j \in J} \sum_{i \in I_j} \left| l^i_j - \hat{l}^i_j \right| / l^i_j
\]  

(5)

where \( J \) denotes the set of the total jobs, \( I_j \) denotes the collection of configurations for job \( j \), \( l^i_j \) denotes the true value of the latency under the \( i^{th} \) configuration in job \( j \), \( \hat{l}^i_j \) denotes the predicted latency under the \( i^{th} \) configuration in job \( j \), and \( m \) is the number of all the traces. We call a training setting \( LO_j \) where we “leave out” job \( j \) for testing and use other jobs traces for training. We define two types (T1 and T2) of errors on the same test job. For T2 error, the model is trained only on the traces of other jobs, while for T1, the model is trained also on 20% of the traces of the test job. Thus, the T1 error reflects the generalization power over unseen configurations of familiar jobs, while T2 error reflects the generalization power over unseen jobs.

To evaluate T2 error, we do multiple experiments with each regarding one regular workload as the test data set and its corresponding LO data as training and validation set. We compute their average T2 error values as the final T2 error. In particular, we used 10% data from the LO set to validate.

To evaluate T1 error, we also do multiple experiments with each regarding the 80% part of one regular workload as the test data set and its corresponding LO data together with the rest 20% part of the regular workload as the training and validation set. We compute their average T2 error values as the final T2 error. We used 10% data from the LO set to validate as well.

4.2. Current Result

For now, my result is still not good enough even though I have tried a lot of different hyper-parameters for the AE model and the NNR. The current T1 error and T2 error in the data has shown in the following Table 1.

<table>
<thead>
<tr>
<th>SQL-like workload</th>
<th>ML workload</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 error</td>
<td>T2 error</td>
</tr>
<tr>
<td>36.79%</td>
<td>35.94%</td>
</tr>
</tbody>
</table>

Table 1. Current Results

There are at least two weird results, which I think should be caused by overfitting. One is that the overall error is too high, even comparing with the normal sense. The other thing is that T1 error should be lower than T2 error since it includes more training data which has strong relation with the test set. From the experiment, I reckon it was caused by overfitting – the switch between underfitting and overfitting in the experiment is pretty sensitive when I gently changed some hyperparameters.

4.3. Analysis

In this subsection, I will show the details of my current neural network structures, and analyze the problems and potential approaches that I am going to explore in the future.

4.3.1 Network Structures

Our data set has observation vectors with 150 dimension, configuration parameter vector with 10 dimension and one
latency metric as the user objective.

For the AE model, we choose the model as follows, where the fc1, fc2, fc3 and fc4 have the shapes \(150 \times 32\), \(32 \times 16\), \(16 \times 32\), \(32 \times 150\) respectively. We encode workload in 6 dimension (bottleneck layer’s dimension minus the dimension of configuration). If the capacities of hidden layers are larger, the results will be much easier to overfit the result.

Encoder:
input -> fc1 -> leaky-relu -> bn -> fc2
   -> leaky-relu -> z
Decoder:
z -> fc3 -> leaky-relu -> bn -> fc4
   -> sigmoid -> output

For the NNR model, we choose a standard neural network with 4 hidden layers. The input has 16 dimension where we concatenate the workload encoding and the real configuration. The output is one scalar value meaning the latency. Each hidden layer in NNR model has 100 neurons. We present our model as follows.

Input -> (fc -> dropout -> leaky-relu
   -> bn) x 4 -> output

4.3.2 Model Tuning Approaches

Training two models together is a hard work. They are not independent and complex. Basically, our strategy is to fix one model and use grid search to find a temporary best hyperparameter for the other model until the temporary best result for the two models both converge. Followings are the details.

1. **Initial tuning.** We first tuning AE model from scratch. we choose the hyperparameters that can achieve the smallest AE loss value in our experiments as our temporary best AE hyperparameters.

2. **Finding NNR hyperparameters.** We fix our temporary best AE hyperparameters and find the best hyperparameters for NNR model that reaches the lowest MAPE by grid search. The found hyperparameters are regarded as the temporary best NNR hyperparameters.

3. **Finding AE hyperparameters.** We fix the our temporary best NNR hyperparameters and find the best hyperparameters for AE model that reaches the lowest MAPE by grid search. We regard the found parameters as the temporary best AE hyperparameters.

4. **Stop tuning.** We finish tuning until AE and NNR’s the temporary best hyperparameters both converge. Otherwise, we repeat the step 2 and step 3.

4.3.3 Technique Details and Overfitting

During our experiment, we found the models were easy to have overfitting. We applied multiple techniques to reduce overfitting. Such as adding dropout layers, reducing the capacity in the network, using tied weight for the AE model, and adding the weight decay value in the Adam optimizer.

However, the overfitting is still hard to avoid without decreasing the accuracy. Even worse, the distribution of the training/validation data set sometimes has big gap with the testing set. Take the initial tuning as an example. We get the temporary best AE parameters with learning rate 0.03, weight decay 0.001 and \(\lambda = 1\) after validation. As we can see in the figure[4], although the loss of training and validation has a nice look, the test loss is very unstable and sometimes becomes pretty high.

In the future, we plan to try following things to make it better.

1. **Collect more data.** Currently, we only have 4800+ data points, which are very sparse. On the one hand, a larger data set can help improve the robust of the test performance, and at the same time, avoid overfitting and reduce the MAPEs. On the other hand, with more data, we have much more possibility to avoid the distribution gap between the train/validation data and test data.

2. **Try early stopping technique.** Early stop in the training model can include more parameters to test the model, which might help find the parameters before the model is overfitting.

3. **Gently adjust model.** We still can try to adjust the capacity in the neurons and the dropout parameters when add other techniques.

5. Conclusion

We provided neural network architectures to automatically learn a predictive model for a user objective. Based on our model, we can provide good system configurations for a new workload to achieve the best user objective. Our experiment is based on a range of Spark streaming workloads. We hope our model can achieve decent accuracy on the user objective and we are pushing the result to a better level.

This project applies the deep learning technologies to the databases area, which is a pretty interesting and valuable work. In the future, we decide to explore it in multiple ways.

1. The configuration usually has multiple parameters thus leads to a high-dimension representation. Hence, searching a good configuration for a workload costs a lot. To speed up configuration recommendation, we
would like to take use of the workload encoding by mapping a new coming workload to its nearest neighbor workload in our training set. Then we borrow the knowledge from its nearest neighbor and avoid searching configurations from scratch. To make the idea work well, we need the distance relation can be preserved when doing dimension reduction from observations to workload encodings. Therefore, including Triplet loss [10] or Contrastive loss [5] can be a good direction to explore.

2. This project only consider one user objective. However, in a real case, a user may consider multiple objectives. For example, a user might want the latency with no more than 3 second as well as the throughput no less than 10G/s when doing some analysis. Considering trade-offs among multiple objective can be an interesting direction to explore.

3. Currently, we fix the hardware settings and the running environment to run all the workloads. However, in a real case, users may want to choose the most efficient running platform with monetary cost as small as possible, such as a light cluster efficient enough for the target workload. Therefore, taking the running environment into consideration can be a valuable practical future work as well.

From this project, I learned a lot from not only the deep learning techniques themselves, but also tuning multiple models from scratch. Throughout the courses and my own experience on this project, I have a much better understanding on the deep learning techniques, and hope to use deep learning techniques to refresh the problems in databases.
References


