SCAVENGER – A Framework for Efficient Evaluation of Dynamic and Modular Algorithms

Andrey Tyukin , Stefan Kramer, and Jörg Wicker(✉)

Johannes Gutenberg-Universität Mainz, Staudingerweg 9, 55128 Mainz, Germany
tyukiand@students.uni-mainz.de, {kramer,wicker}@informatik.uni-mainz.de

Abstract. Machine Learning methods and algorithms are often highly modular in the sense that they rely on a large number of subalgorithms that are in principle interchangeable. For example, it is often possible to use various kinds of pre- and post-processing and various base classifiers or regressors as components of the same modular approach. We propose a framework, called SCAVENGER, that allows evaluating whole families of conceptually similar algorithms efficiently. The algorithms are represented as compositions, couplings and products of atomic subalgorithms. This allows partial results to be cached and shared between different instances of a modular algorithm, so that potentially expensive partial results need not be recomputed multiple times. Furthermore, our framework deals with issues of the parallel execution, load balancing, and with the backup of partial results for the case of implementation or runtime errors. SCAVENGER is licensed under the GPLv3 and can be downloaded freely at https://github.com/jorro/scavenger.

1 Introduction

Consider the following example: Suppose we want to compare different instances of a modular algorithm $A_{f,g}$ that consists of two major parts: a preprocessing part f, and a core algorithm g: $A_{f,g}(x) = g(f(x))$, with $f \in \{f_1, \ldots, f_n\}$ and $g \in \{g_1, \ldots, g_m\}$. We could evaluate each combination separately, and then choose the best performing combination. However, this results in a repeated computation of each $f_i(x)$, one time for each possible core-algorithm g_j. Hence, it is beneficial to cache and to keep intermediate results, and to share these results between the members of the algorithm family $\{A_{f,g}\}_{f,g}$ to avoid unnecessary recomputation. Yet, we want to keep parallel execution, although the evaluation of the family of algorithms with shared subalgorithms is no longer *embarrassingly parallel*.

In the demo, we present SCAVENGER, a framework that simplifies the representation and evaluation of families of algorithms that share common subalgorithms. The user of SCAVENGER formulates a family of algorithms in a declarative style as a directed acyclic graph (DAG) of subalgorithms which can in turn spawn arbitrarily complex DAGs to carry out its computation. The whole job, represented as a DAG is then evaluated in a way that is suitable for single or

A. Bifet et al. (Eds.): ECML PKDD 2015, Part III, LNAI 9286, pp. 325–328, 2015.
DOI: 10.1007/978-3-319-23461-8_40

multiple computers (including clusters). In contrast to cluster engines or map-reduce approaches that process large amounts of data with a single algorithm, our framework is focused on the use case with a moderate amount of data, but a large number of algorithms.

Workflow systems like Taverna[1], Knime[2], WEKA[3], Pipeline Pilot[4], or ADAMS[5] provide an easy-to-use interface to carry out experiments with repetitive steps. This makes them closely related to Scavenger. Yet, caching and reusing of intermediate results is not a central aspect of these systems, which is the core functionality and benefit of Scavenger. Additionally, Scavenger was designed with an easy-to-use mechanism to run on clusters in contrast to workflow systems, which tend to orient more on the single desktop use case (note that we talk about the emphasis of the system, exceptions do exist).

2 The Scavenger Framework

The back-end of the Scavenger is built on top of the Akka framework[6]. Akka provides an implementation of an actor system that can be distributed across multiple physical compute nodes connected by a network. Actor systems are hierarchical collections of actors. Actors are lightweight entities that are characterized by their internal state and their reactions on incoming messages. Every actor has its own mailbox, and communicates with the outside world exclusively by sending and receiving messages. Conceptually one can think that each actor is executed on its own thread, however, in reality actors are much more lightweight than threads of the operating system. The messages are usually simple, immutable, serializable JVM objects which can be either passed by reference within a single JVM or serialized and sent over network via the TCP/IP protocol.

The Akka framework ensures that all actors get enough CPU time for the whole system to stay responsive, delivers the messages sent by actors, handles error propagation within the actor system, and also takes care about the communication between multiple actor systems that are running on different physical machines. Users have only to specify various types of actors with their behaviors as well as the kinds of messages that are exchanged between the actors.

The Scavenger back-end uses multiple types of virtual nodes in order to do its job. Each node is controlled by an actor that is responsible for communication with other nodes. For simplicity, one can think of each virtual node running on a separate physical machine, but this is actually not required: multiple Scavenger nodes can coexist on the same physical machine, or even be executed within a single JVM. The back-end implements the classical Master-Worker pattern. There are currently three types of nodes: the seed node, the master node, and

[1] see http://www.taverna.org.uk/
[2] see http://www.knime.org/
[3] see http://www.cs.waikato.ac.nz/ml/weka/
[4] see http://accelrys.com/products/pipeline-pilot/
[5] see https://adams.cms.waikato.ac.nz/elgg/
[6] see http://akka.io

the worker node. When a Scavenger service is running, a single seed node, a master node, and multiple worker nodes must be active.

The seed node is responsible for establishing connections between all other nodes. The main purpose of the seed node is to wait for a handshake message from the master and then to tell each worker node where the master node is.

The master node communicates directly with the client application. It translates the incoming computation-valued requests from the client into a form that is suitable for parallel computation by subdividing it into smaller tasks, and then schedules these tasks for execution. It coordinates the work of the worker nodes, and is responsible for load balancing and dealing with failure of single worker nodes. It is also responsible for caching and backing up of intermediate results.

The worker nodes provide the raw computing power. They receive internal jobs, compute the result, and send it back to the master node.

3 Case Study: Autoencoders

Autoencoders are neural networks that can be trained in an unsupervised manner [1]. The input and the output clamped to both ends of the autoencoder are the same. The information is pressed through the central bottleneck layer. The activations of neurons in the central layer can be considered a compressed representation of the data. We train autoencoders as follows. We start with a single layer of neurons. Then we keep unfolding the innermost layer, eventually train every new layer separately, and then tune the whole network using backpropagation. Furthermore, we have the possibility to train the final inner layer as an Restricted Boltzmann Machine (RBM) or to tune it as a separate autoencoder with one single layer. We can also tune the whole network using backpropagation. The trained network can have different depths and different dimensions of layers. Obviously, if one autoencoder arises as an unfolded version of another autoencoder, models will share some common intermediate results when executed. We want to briefly sketch how one would approach the training of the family of autoencoders using the Scavenger framework. First, we have to define the atomic algorithms `unfold`, `trainInnerAsRbm` `tuneInnerAsAnn`, `tuneWholeNetwork`:

```
1  case class TuneInner extends Algorithm[(Data, Autoencoder), Autoencoder] {
2      def identifier = formalccc.Atom("tuneInner")
3      def difficulty = Parallel
4      def cachingPolicy = CacheGlobally
5      def apply(dataAndEnc: (Data, Autoencoder), ctx: Context):
6      Future[Autoencoder] = {...}
7  }
```

The implementation of iterative optimization methods requires some more work. The reason is that we have to transform synchronous `while` loops into asynchronous code. For this, the Scavenger framework provides asynchronous control structures like `async_while`. In an iterative optimization method like backpropagation, we have to replace all `while` that have a body that requires substantial computation time by `async_while`. Furthermore, we have to chain asynchronous calls by `Future`s `flatMap` instead of usual semicolons:

```
case class Backpropagation extends Algorithm[(Data, Autoencoder), Autoencoder] {
    ...
    def apply(da: (Data, Autoencoder), ctx: Context):
    Future[Autoencoder] = {
        async_while(predicate) {/* submitting ad-hoc jobs to 'ctx', waiting for
             results*/}.flatMap{...}
```

When all the basic atomic algorithms are defined, we might want to partially apply some of them to the data, so that we have a uniform collection of algorithms that map `Autoencoder`s to new `Autoencoder`s, and do not require any extra input:

```
val data = Computation("theData") { /* load data from file */ }
val tuneWholeWithData = tuneWholeNetwork.partialFst(data)
```

Now suppose that we have an implementation that trains an autoencoder with a specified training strategy:

```
def autoencoder(trainingStrategy: Algorithm[Autoencoder, Autoencoder],numLayers):
     Computation[Autoencoder] = {...}
```

Since we do not know which combination performs best according to some error measure, we generate every possible combination of `trainInnerAsRbm`, `tuneInnerAsAnn` and a identity operation `Id`:

```
val strategies = for {
    f <- List(trainInnerAsRbmWithData, Id)
    g <- List(tuneInnerAsAnnWithData, Id)
    h <- List(tuneWholeWithData, Id)
} yield (h o g o f)
```

Now we submit multiple autoencoder-jobs to the SCAVENGER framework. We vary the depth and the training strategy.

```
val futures = for {
    s <- strategies
    n <- List(2,3,4,5)
} scavengerContext.submit(autoencoder(s, n))
val allResults = Future.sequence(futures)
val results = Await.result(allResults)
```

After this, depending on how we are executing the application, we could issue some cleanup instructions and shut down the system.

4 Conclusion

Our framework combines the aspects of caching, persistence of intermediate results, and parallelism. It provides a simple and minimalist user API and can be used on single computers or clusters. It can simplify and accelerate the development and execution of experiments that involve a large number of algorithms that share common subalgorithms. In the demo, we will show the simplicity and power of the framework by means of simple examples that will be developed step by step.

References

1. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science **313**(5786), 504–507 (2006)