Machine Learning as a Reusable Microservice

Marc-Oliver Pahl
Technical University of Munich
pahl@net.in.tum.de

Markus Loipfinger
Technical University of Munich
m.loipfinger@tum.de

***Abstract*—Machine Learning is recently becoming a universal problem solving tool. However, implementing machine learning (ML) into applications is difficult, time intense, and requires expert knowledge. We encapsulate machine learning as a data-oriented microservice that can simply be used to mash up applications with machine learning capabilities. To illustrate the approach we identify three machine learning algorithms that are relevant for the Internet of Things (IoT): Feed-Forward Neural Networks (FFNN), Deep Believe Networks (DBN), and Recurrent Neural Networks (RNN). We analyze those algorithm's characteristic properties and model them as configurations for dynamically linkable REST ML service modules. Our approach strictly separates the algorithm implementation from its configuration. It allows a simple extension with diverse ML algorithms. Following a service oriented design, we implement the training of our neural networks as a separate module. We evaluate how the performance of our solution compares to directly programming the chosen TensorFlow library. Our approach facilitates the implementation of ML-based data analytics significantly by enabling reuse and sharing of executables and configurations. It enables rapid prototyping and an explorative use of ML.**

***Index Terms*—machine learning, neural networks, service orientation, microservices, encapsulation, reuse, high usability**

I. Introduction

Google's neural network "dreams" made a broader audience aware of the capabilities of Machine Learning (ML) and its subset Artificial Neural Networks (ANN) in 2015[1]. Since then, ML emerged as commonly applied data analytics technique in many science fields[2] and industry domains[3].

A major problem with applying ML today is *usability*. Getting the necessary background, and understanding the tools are *time consuming* and *complex*. Both require expert knowledge from service developers. Major companies recognized this problem and offer easy-to-use Application Programming Interfaces (API) for *specific problems* such as text recognition, face recognition, etc. (Sec. II). However, such APIs have two significant drawbacks:

- <C.1> They are *tailored for specific applications* and *cannot easily be extended*.

This limits the creativity of developers by channeling their workflows to the libraries predefined ones.

- <C.2> They are typically *cloud-based*.

This is problematic for other architectures such as edge-computing, which is likely to be found in many Internet of Things (IoT) installations in the future, and for mobile computing where Apps are may require working offline without connectivity to a cloud. When providing ML outside the cloud, a big advantage besides lower latency and connectivity requirements, and higher bandwidth is *data privacy*. Cloud based approaches require uploading data to cloud providers, which may conflict with privacy requirements.

In addition to cloud based offers, commercial vendors and open source projects offer ML libraries that facilitate a fully-local implementation of ML (see Sec. II). However, these APIs also have two significant drawbacks:

- <C.3> They are often *complex* to use since they are <C.3.a> *heterogeneous*, and they <C.3.b> *lack intuitive, explorative workflows* that would ease their use especially for inexperienced developers.
- <C.4> They are not built for a executable and configuration *reuse and sharing between multiple applications*.

While enabling to save and exchange configurations, the service oriented features *reuse* and *sharing* are not enough covered by the state of the art. Sharing implementations and exchanging configurations of ANNs can significantly leverage the cross-vendor integration of ML into *different* applications. It can also save computing resources on a computing node.

This paper introduces ML as a microservice that is standalone, locally runnable and simply reconfigurable. We address the four identified challenges: we show how <C.1> generic ML libraries can be encapsulated as REST services with <C.3.a> a unified interface that can run <C.2> without cloud back-end. We show how our *customization by-configuration-only* approach allows an <C.3.b> explorative configuration parameter refinement, resulting in good results even for inexperienced ML novices. We discuss how our solution <C.4> enables sharing configurations and binaries among multiple applications.

Coming from the IoT, we exemplify our approach at ML algorithms that we identified as most suitable for our domain: Feed-Forward Neural Networks (FFNN), Deep Believe Networks (DBN), and Recurrent Neural Networks (RNN). Though exemplifying our concept at those algorithms, our implementation and our methodology can easily be adapted for other ML algorithms.

In contrast to many existing libraries that are function-call centric our approach is *data-centric*. At practical examples we

This research has been supported by the German Federal Ministry of Education and Research (BMBF) in the project DecADe (16KIS0538).

[1]https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html

[2]https://trends.google.com/trends/explore?date=all&q=machine%20learning

[3]http://www.gartner.com/newsroom/id/3412017

show how this additionally facilitates the use of our approach.

In Sec. II we present libraries that facilitate the use of ML. We also discuss the APIs offered by major cloud providers. In Sec. III we identify and briefly introduce our IoT example ANNs: FFNN, DBN, and RNN. We identify common, and algorithm-specific characteristic configuration parameters.

Sec. IV introduces our encapsulation of ML as a REST service with a *strict separation between algorithm and configuration*. For the configuration we use the identified parameters from Sec. III. We present how *learning can be implemented as a separate service*. In the evaluation (Sec. V) we compare the performance of directly using the TensorFlow library to our encapsulation approach for both, the learning phase and the running phase. The increased usability for developers is assessed by implementing a standard ML benchmarking problem. We also show how our approach enables improving ANN configurations iteratively.

II. State of the Art

The relevance of facilitating the access to ML algorithms is known. Different initiatives target making ML algorithms more accessible for developers. In the following we present approaches that we consider most relevant.

Three key differences to our work are 1) that none of the state of the art approaches offers a *fully encapsulated, REST interface* (<C.3> (✗)). Our approach is *data-centric* while the identified state of the art is function-centric. It requires calling library functions while our service is only configured via parameter tuples and fed data for processing. Our approach facilitates the use and *enables flexible mash-ups*.

Also 2) none of the approaches allows to *exchange configurations and share runtimes with other developers as easy* as our solution does (<C.4> ✗). This again facilitates the use and *reduces the setup time and complexity for developers*. The shared runtime use *saves resources*, which is especially relevant in the IoT.

Finally, 3) we designed our solution to *run locally* at the edge of the Internet. Only some state of the art offers this (<C.2> (✓)). Local processing is especially relevant in the IoT for providing *privacy preserving data processing*. Many of the related work is cloud-based and therefore violates this goal. Even if privacy is not an issue and a cloud based approach is chosen, we consider applying our concepts beneficial for the usability reasons given before.

Our concept is compatible with the state of the art frameworks. Integrating them is straight forward. It would extend our implementation with many more ML algorithms.

Scikit-Learn [1] is a Python library. Its developers also recognized the need for a simple-to-use ML API. The website is a great resource for getting an understanding of different ML algorithms and their effects on data. This complements our effort to facilitate the use of ML. Integrating the Scikit back-end would extend our pool of algorithms.

Already in 1992 the WEKA [2] project started. It provides a great interface for exploring diverse ML algorithms. An interactive explorer like the one offered by WEKA is complementary to our effort and could be implemented as another service in the future to facilitate the use of ML for inexperienced developers even more.

Apache Mahout [3] provides a set of ML algorithms. The library fits especially well for big data analysis as part of it can run in a Hadoop cluster enabling efficient parallel processing of big amounts of data. Integrating Mahout with our interface would extend our set of offered ML algorithms.

So far WEKA and Mahout seem not to support the ANNs we identified as most relevant for the IoT. However, like our solution both can be extended (<C.1> ✓).

There are several offers by commercial cloud providers (<C.2> ✗): Google[4], Amazon[5], Microsoft[6], and to some extent IBM[7] offer APIs for common services such as text or image recognition. Their frameworks offer highly specialized functionality (<C.1> ✗). Some solutions offer the possibility to share fully configured algorithms with other developers, not delivering our flexibility but fulfilling the goal (<C.4> (✓)).

In addition to the specialized interfaces, most cloud providers also offer low-level interfaces to their ML back-ends. Google offers its TensorFlow framework[8]. We use TensorFlow for our implementation. Microsoft offers the Cognitive Toolkit[9]. Amazon offers installations of the former two plus linear learning algorithms[10].

Typically those general interfaces offer most of the ML functionality of the cloud offers (<C.1> (✓)) but are not bound to the cloud. However, not all vendors offer their technology for offline use (<C.2> (✓)). The implementation of ML using the bare library APIs is more complex than our REST interface (<C.3> (✗)). It requires programming, configuring, and training the ANN. The sharing of configurations is possible but not in the focus of the libraries (<C.4> (✓)).

III. Machine Learning in the Wild

We want to encapsulate ML as a *microservice with a REST interface* that can be used and *customized* for diverse applications only by *changing structured configuration parameters*. This requires decomposing ML into fixed parts that can be reused and dynamic parts that have to be customized for concrete applications.

ML is a broad field with many different algorithms. As we come from the IoT domain, we want to exemplify our concept with ML algorithms that suit IoT use cases. By surveying several IoT scenarios we identified three ML algorithms that are frequently used. The chosen algorithms are Feed-Forward Neural Networks (FFNN), Deep Belief Networks (DBN), and Recurrent Neural Networks (RNN).

FFNNs are the most simple ones of our selection. Their typically use is classifying things such as events. In 2010, [4] used FFNN for automated lighting control.

[4]https://cloud.google.com/products/machine-learning/
[5]https://aws.amazon.com/machine-learning/
[6]https://azure.microsoft.com/en-us/services/cognitive-services/
[7]https://www.ibm.com/watson/, commercial focus.
[8]https://www.TensorFlow.org/
[9]https://azure.microsoft.com/en-us/services/cognitive-services/
[10]https://aws.amazon.com/amazon-ai/amis/

They are often used for preprocessing data before fitting it into other algorithms. In 2013, [5] used DBN for human behavior prediction. Also in 2014, [6] used DBN with Restricted Boltzmann Machines (RBM) to recognize activities in IoT smart spaces.

RNNs finally enable taking state such as past events into account. They can process sequential events such as human actions. In 2011, [7] used FFNN and RNN to detect human behavior patterns. In 2014, [8] used FFNN and RNN to develop a support system for ambient assisted living.

We have a closer look at FFNN, DBN, and RNN. All three implement deep learning, meaning that their networks consist of many "deep" layers of neurons. We identify their characteristic parameters to factorize them out of the shared microservices we want to offer for each of the three algorithms.

Regarding the configuration of ANNs, *hyperparameters* and *parameters* are relevant. *Hyperparameters* adjust the fundamental layout and behavior of an ANN. Developers define aspects such as the architecture and dimensions of an ANN, or how an ANN is trained. *Parameters* define the concrete configurations of the neurons within an ANN. They are the result of training a neural network in a so-called *learning phase*.

For improving the quality of the output of an ANN both, the hyperparameters and the parameters can be adjusted. Visualizations such as those provided in Scikit-Learn [1] and WEKA [2] can help getting a more intuitive understanding of the changes. Our proposed solution makes such changes very easy as it simply consists of changing values of parameters in the configuration data (see listing 1). In Sec. V-D) we discuss how our solution can be used to explore an optimal configuration of the hyperparameters of an ANN.

A. Neural Network Foundations

an ANN consists of interlinked neurons. For processing, each neuron receives an input vector $\vec{x}$. During processing, among other steps, the input vector components are weighted with w and a bias b is added. The outcome y of a neuron is obtained by feeding the result into an activation function a.

$$y = a\left(z\right) \tag{1}$$

$$z = \sum_i w_i x_i + b \tag{2}$$

Different neurons differ in their activation functions. Three frequently used neurons are the *Sigmoid Neuron* ($a(z) = \sigma(z)$), the *Tanh Neuron* ($a(z) = \tanh(z)$) , and the *Rectified Linear Neuron* ($a(z) = \max(0, z)$). Each neuron can be characterized by the parameters a, w, and b. The amount of neurons and their connectivity characterize ANNs.

B. Learning Approaches

Learning is used to iteratively configure each neuron of an ANN. The learning phase results in the final *parameters* of an ANN. By providing the intended outcome of a classification network to the trainer (supervised learning), or by letting it guess it (unsupervised learning), a neural network learns which configuration parameters fit best.

While learning, an ANN iteratively improves the w and b of the neurons by applying a cost function C. C is also referred to as loss function (Sec. V). Another relevant parameters for learning is the *learning rate* η. η defines how much the adaptation in the current step is considered; in other terms 'how fast' the used algorithm learns.

During the learning phase, the trainer goes through the training data set multiple times. Each time is called training epoch. A training epoch typically consists of multiple mini-batches [9]. m indicates the size of the mini-batches, which are used to segment the training data for making the processing more efficient.

C. FeedForward Neural Networks

Multi-layer *FeedForward Neural Networks* (FFNNs) (or *Multilayer Perceptrons* (MLPs)) consist of one input layer that processes the input data, one to multiple hidden layers, and one output layer that creates the output. Information travels only in one direction from the input to the output layer. The neurons are fully-connected, meaning that each neuron in one layer is connected to each neuron in the next layer. When many hidden layers are used, a FFNN is called *deep*.

D. Deep Belief Networks

Deep Belief Network (DBN) are composed out of simple ANN that are organized in layers. Only the layers are connected. The neurons within layers are not connected. In our pilot we use *Restricted Boltzmann Machines* (RBMs). Each RBM consists of two layers. The hidden layer of RBM_{n-1} is used as visible (input) layer of RBM_n. During training all parameters of the RBMs are iteratively trained.

For training the described DBN, the RBMs are trained individually layer by layer.

E. Recurrent Neural Network

FFNNs and DBNs use a fixed set of input nodes. *Recurrent Neural Network* (RNN) can process inputs of arbitrary length. In contrast to the other networks they possess feedback loops. Later input is processed taking previous input into account resulting in context-dependent results.

Today, popular RNNs use *Long Short-Term Memory* (LTSM) or *Gated Recurrent Units* (GRUs) as hidden units s. RNNs are trained similar to FFNNs. However, their statefulness allows specifying additional parameters.

IV. Machine Learning as a Service

We propose machine learning as a μS. By this we understand providing machine learning algorithms encapsulated in REST μSs that can be customized by changing predefined, easy-to-use, and service-external, well-structured configurations instead of programming the desired functionalities.

We target enabling a mashup of microservices (μS) for realizing complex scenarios such as the IoT scenarios from Sec. III. To implement the envisioned μS oriented design,

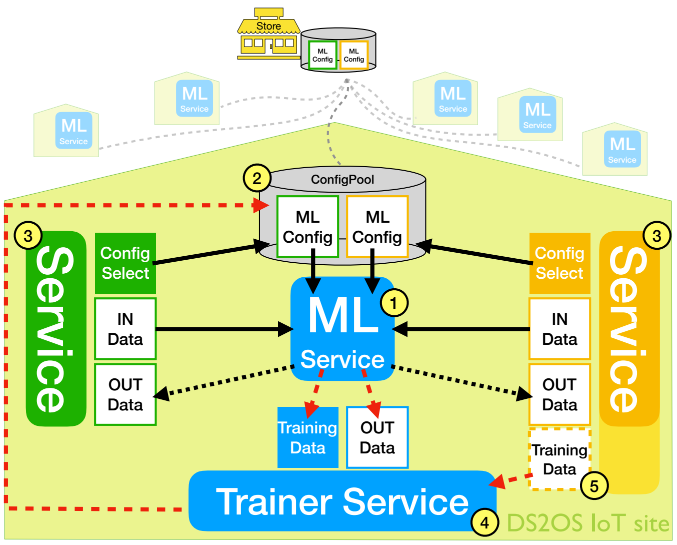

Fig. 1. ML as a μS with external configurations.

we use the Distributed Smart Space Orchestration System (DS2OS) with its Virtual State Layer (VSL) middleware back-end [10]. The VSL is a self-organizing peer-to-peer system that offers a distributed blackboard.

The VSL facilitates the development of μSs by providing developers with functionality for *storing* and *exchanging* μS *data* as well as *discovering* other μSs based on semantic attributes and data-centric μS *coupling* (late binding) over its blackboard tuple space [11]. Thereby it forms the base for implementing (complex) services as μS mashups in a Service Oriented Architecture (SOA) [10].

The VSL uses key-value tuples with different additional properties for organizing its data nodes in the blackboard. Hierarchical compositions of node structures are globally shared over a store that is also used to distribute μSs such as the μSs we present in this paper [12]. We use this feature to distribute ML configurations globally.

In Fig. 1 all μSs within a DS2OS site –which can be fully local or have parts in the cloud– use the VSL for data storage and couple over it. On the top of Fig. 1 the DS2OS store is shown that is used for globally exchanging data between sites such as μSs and their configurations.

VSL μSs offer data over a REST interface. The VSL access methods are get, set, subscribe and notify. When a *μS1* wants to read access data of a *μS2*, it executes "get /vsl/μS2/value23".

Conceptually our ML solution could also be implemented using plain text files for the configurations and plain web-service REST access, e.g. via an HTTP GET. However, by covering all aspects of μS coupling, composition, and data exchange and management, the VSL strongly supports our goal to offer ML as a reusable and easy-to-use μSs (<C.3>) and is therefore used.

Fig. 1 shows the proposed architecture. Its key concept is a full separation of the ML μS (1) that implements a ML algorithm, and its configuration (2). The ML μS executable can be used by multiple μSs (3) (<C.4>) as it is stateless and only customized to the requirements of each μS (3) via the configuration. By having the configurations available in a configuration pool (2), μSs can easily invoke the ML μSs with a certain existing configuration, enabling the sharing of configurations among μS (<C.4>).

For the configurations we use all parameters (hyperparameters and parameters) that are characteristic for the ML algorithm implemented in the ML μS (Sec. III). Developers can *fully* customize an algorithm simply by changing the configuration parameters (<C.1>).

We use the VSL key-value pairs to represent the configurations (ML Config). The VSL context models allow setting standard values for all parameters. That enables developers to configure only those parameters they want to change. For a concrete ML Config (2), a developer can simply set the VSL addresses of the nodes the ML μS should use as input and output data.

```
1  <MLConf42 type="FFNNConfig">
2    <InputDataAddresses>
3    <node>/vsl/agent23/svc4/sensor1/motion</node>
4    [...]
5    <node>/vsl/agent42/svc375/sensor123/motion</
          node>
6   </InputDataAddresses>
7   <OutputDataAddresses>
8     <node>~/presenceDetected</node>
9   </OutputDataAddresses>
10 </MLCOnf42>
```

Listing 1. A VSL context model representing a ML Config.

Listing 1 shows the corresponding VSL data in XML representation. The configuration *MLConf42* inherits its default parameters from the VSL context model *FFNNConfig* (line 1). It overwrites the data nodes *InputDataAddresses* and *OutputDataAddresses* with the values that are suitable for the specific setup (lines 2-9). The input addresses represent nodes in other μSs (svc4, svc375). The output address is in the VSL address space of the ML μS. The output is written into the subnode *presenceDetected*. The listed ML Config configures a ML μS to run an FFNN on input data nodes at addresses */vsl/agent23/svc4/sensor1/motion, etc.*, and to write a single output into the local variable *presenceDetected*.

When a suitable ML Config (2) exists already, μS developers only have to inform a ML μS (1) of the data they want to have processed, the desired location of the output, and the intended ML Config (3). This makes the use of ML as a μS possible with few lines of code and low time investment (Sec. V, <C.3>).

```
public static void main(String[] args) {
 final mSConnector connector = new
      ServiceConnector(agentUrl, keystore,
 "password");
 connector.activate();
 connector.set("vsl/agent42MLs/addConfig", "vsl/
      path/MLConf42");
 connector.subscribe("vsl/agent42MLs/
      presenceDetected", callBackFunc);
```

Listing 2. A VSL μS that adds the MLConfig42 to the ML μS.

Listing 2 shows the entire Java code of a VSL service that invokes of a ML μS with the configuration stored at the

VSL address *vsl/path/MLCOnf42*. After being configured the ML μS (1) subscribes to the defined input VSL data nodes (lines 3-5 in listing 1). On input node change, the ML μS gets notified, computes its output, and stores it in the VSL output node(s). All μS with VSL access to this value can consecutively use the new data. They can subscribe to the node *vsl/agent42MLs/presenceDetected* and fetch the output once being notified of its availability.

The configuration is passed from the μS (3) to the ML μS (1) with line 5 of the Java code shown in listing 2. As can be seen from the listings 1 and 2, in combination with the mechanisms provided by the VSL our approach *simplifies the use of ML within a μS significantly* (<C.3>).

Following our μS oriented approach we also factorize the learning out of the ML μS. See ML μS (1) and Trainer μS (4) in Fig. 1. The ML μS implements the ML algorithm. It is *stateless*. All state is stored in the configuration in the VSL (ML Config, 2), and dynamically loaded when the corresponding ML functionality is requested.

The Trainer μS (4) implements the functionality to train the ANN offered by the ML μS (1). Training data (5) is handed over to the Trainer μS (4) by other μSs (5). The Trainer sends the training data to the stateless ML μS (1), It evaluates the output of the ml μS, and updates the corresponding parameters in the ML Config (2) accordingly (Sec. III-B).

These are all steps necessary for training the ANN in the ML μS (1). The training results in a new ML Config (2). It matches the needs of the calling μS (3). It contains all state (hyperparameters, parameters) specializing the ML μS (1).

Separating ML μS (1) and Trainer (4) makes sense as both offer different functionality. Depending on the ANN it could make sense to offer different trainers or even combining training approaches. Our approach facilitates both.

Having both services stateless fits as the specialization comes via the μSs using the ANN functionality (3) and not out of the ANNs themselves. Such specialization happens either by configuring the hyperparameters identified in Sec. III, building upon existing configurations, or using the default values. The resulting configuration are directly usable by other services.

The possibility to work with few parameters, and having explanations *facilitate the work for both, ML novices and experts* (<C.3>). For the latter the handling of the ML libraries becomes significantly more efficient (Sec. V). The reduction of the configuration to few parameters enables an explorative configuration of ANNs by iteratively changing parameters and evaluating the result (Sec. V-D, <C.3.b>).

The VSL provides full location transparency. This allows to deploy the ML μS on any computing node within an IoT system. It can thereby run in the cloud or at the edge (<C.2>). Cross-site sharing of ML Configs happens over the DS2OS store. See Fig. 1 on top.

V. Evaluation

We implemented the ML and training μSs for FFNN, DBN, and RNN using the Google TensorFlow Python library. As TensorFlow implements all three ANNs, for all evaluations we use the same ML μS with different configurations.

We provide a default configuration for each ANN (Sec. IV). Consequently developers only have to specify the parameters they want to change from the default.

Our ML μS is stateless. It has to reload its parameters each time it is called. We expect additional latency. Therefore we compare the *performance* of directly calling the TensorFlow library and our ML as a Service (MLaaS) approach.

For being able to compare the three algorithms, and to enable comparison with other works, we use a common benchmarking for ML algorithms: the classification of handwritten digits using the MNIST data set digits[11].

Regarding *usability* we compare the implementation duration, and the lines of code using the standard library and our approach. Finally we show how our configurations can help tuning the hyperparameters of an ANN iteratively.

A. Runtime Performance

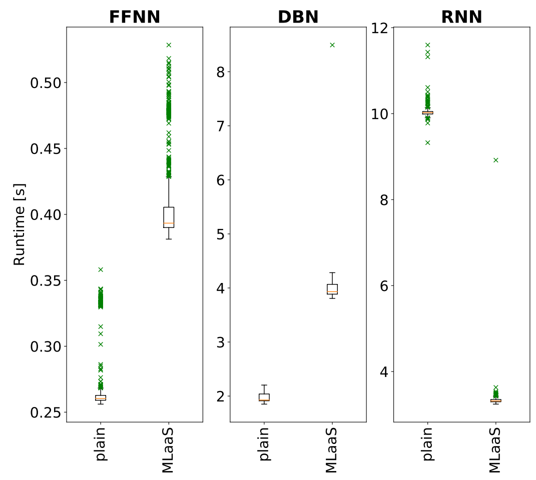

Fig. 2. Performance comparison in s between directly using the TensorFlow library and our proposed ML as a Service (MLaaS) for all three algorithms.

Fig. 2 shows the results for classifying the MNIST dataset for each algorithm. For each plot we processed the entire dataset 1000 times. The boxplots show the quartiles with the 50% quantile as orange line in the middle. Values that are more than 1.5 from the inter quartile range (IQR) are plotted as outliers. On the left side of each of the three plots the latency of using the Tensorflow library directly is shown. On the right the latencies of the same classifications with embedding the ML via our REST encapsulation is shown.

The latency for invoking the ML μS, loading the configuration, passing the parameters, and fetching the result becomes clearly visible. For the FFNN this overhead is about 150ms, and for the more complex DBN it is 2s. This delay could be removed by adding caching to the ML μS so that the last N configurations remain locally cached and only missing configurations are loaded.

[11]http://yann.lecun.com/exdb/mnist/

For the RNN the plot shows a faster runtime for our implementation. The reason is unclear to us. It could be that resetting and restoring the RNN may actually be faster than the library's internal processes.

B. Learning Performance

The processing steps of the Tensorflow library are significantly longer for learning than for classification. When comparing our learning as a service approach with the direct invocation of the library the relative difference can be neglected. This is not surprising as our approach only adds a constant overhead per library invocation.

C. Implementation Performance

For assessing the usability we did a cognitive walkthrough by implementing the digit recognition task used as benchmark before. As before we did so by directly linking the TensorFlow library and using its API, and with our as-a-service solution. Table I compares the implementation times and the resulting Lines of Code (LoC).

TABLE I
IMPLEMENTATION TIMES AND LINES OF CODE (LOC).

algorithm	task	bare lib		our approach	
		t	LoC	t	LoC
FFNN	impl.	5min	85	30s	2
FFNN	setup	2min		30s	
DBN	impl.	8min	148	30s	2
DBN	setup	3min		30s	
RNN	impl.	10min	128	30s	2
RNN	setup	5min		30s	

Our approach significantly reduces the development times for the implementation and the setup for each of the ANNs. The identical numbers on the right side reflect that our structured approach with default values helps unifying the complexity of using any of the ML algorithms.

The listed time for using the bare library is the time we needed when exactly knowing what to implement. In a realistic setting this knowledge is not available. In such cases the differences can be expected to be significantly higher.

For additional validation we implemented the identified IoT scenarios (Sec. III) with our approach in less than 1h each.

D. Explorative Configuration Finding

Our configuration files list the parameters of an ANN algorithm (Sec. III). Any library specific API calls etc. are removed by this abstraction. The structure of the configuration files (listing 1) and the default parameters allow playing around with different hyperparameters easily. Even playing with the hyperparameters by combining different training methods becomes possible. Even without having an interactive GUI like the one provided by WEKA, developers can easily experiment with the hyperparameters and assess their classification results. A GUI could be added as another μS to our solution.

Fig. 3 shows the training of the digit recognition scenario. The x-axis shows the training iterations. The y-axis shows the validation loss that is ideally zero (Sec. III). The lowest curve shows the optimal configuration with a loss of 0.09. The other curves show how the changing of the learning hyperparameters changes the digit recognition quality:

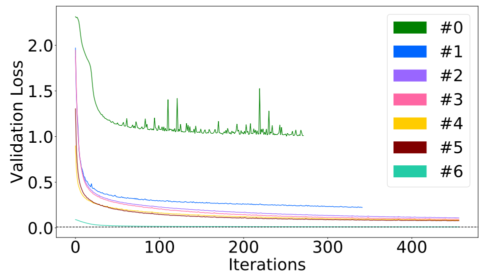

Fig. 3. Iterative hyperparameter optimization. The dotted line (bottom) shows the best loss value reached on the validation set at 0.009.

0) default values; high error value and a high oscillation, which is caused by a high learning rate,
1) decrease of the mini-batch size and the learning rate, weights are initialized randomly; error on the validation set decreases fast,
2) number of hidden units, and the training epochs are increased, and the learning rate is decreased,
3) changes of activation function,
4) decrease of learning rate,
5) change of cost function; good error value.

By providing parameter annotations in the configurations we expect developers to achieve a fast convergence as shown in Fig. 3. The unified configuration files also lay the base towards fully automated configuration parameter refinement.

VI. CONCLUSION

We showed how ML implementations can be modularized to facilitate and speed-up the implementation of ML functionality in services. Our solution enables implementing complex use cases with a service oriented mashup of μSs [10], [13].

We implemented ML algorithms as stateless μSs containing the ML algorithm, and an external configuration that specializes the generic ML service for needs of an application. We showed how the training can be encapsulated in external μSs.

Our evaluation showed that via the structured and annotated configuration files, and automatically provided default parameters the configuration of an ANN becomes very fast and the lines of code get significantly reduced.

We showed that the solution adds latency since the configurations have to be loaded on each call of the ML μS. For many IoT use cases such a delay is not relevant. However, we are working on reducing it and proposed caching therefore.

Our proposed vendor- and developer-comprehensive sharing and exchange of configurations facilitates and speeds-up the adding of ML to service implementations again. It matches the intended service-oriented design.

By simplifying the use of ML significantly we hope to enable more developers to make use of ML.

References

[1] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software - experiences from the scikit-learn project." *CoRR*, 2013.

[2] M. A. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software - an update." *SIGKDD Explorations*, 2009.

[3] R. Anil, S. Owen, T. Dunning, and E. Friedman, *Mahout in Action*, Manning Publications Co. Sound View Ct. 3B Greenwich, CT 06830, 2010. [Online]. Available: http://manning.com/owen/

[4] C. A. Hernandez, R. Romero, and D. Giral, "Optimization of the Use of Residential Lighting with Neural Network," in *2010 International Conference on Computational Intelligence and Software Engineering*. IEEE, 2010, pp. 1–5.

[5] S. Choi, E. Kim, and S. Oh, "Human behavior prediction for smart homes using deep learning." *IEEE International Symposium on Robot and Human Interactive Communication*, 2013.

[6] H. Fang and C. Hu, "Recognizing human activity in smart home using deep learning algorithm," in *Proceedings of the 33rd Chinese Control Conference*. IEEE, 2014, pp. 4716–4720.

[7] A. Badlani and S. Bhanot, "Smart home system design based on artificial neural networks," in *Proc. of the Word Congress on Engineering and Computer Science*, 2011.

[8] A. Hussein, M. Adda, M. Atieh, and W. Fahs, "Smart Home Design for Disabled People based on Neural Networks." *International Conference on Emerging Ubiquitous Systems and Pervasive Networks*, 2014.

[9] M. Nielsen. (2017) Neural networks and deep learning. [Online]. Available: http://neuralnetworksanddeeplearning.com/

[10] M.-O. Pahl, G. Carle, and G. Klinker, "Distributed Smart Space Orchestration," in *Network Operations and Management Symposium 2016 (NOMS 2016) - Dissertation Digest*, 2016.

[11] M.-O. Pahl, "Data-Centric Service-Oriented Management of Things," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, Ottawa, Canada, May 2015, pp. 484–490.

[12] M.-O. Pahl and G. Carle, "Crowdsourced Context-Modeling as Key to Future Smart Spaces," in *Network Operations and Management Symposium 2014 (NOMS 2014)*, May 2014, pp. 1–8.

[13] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2016, pp. 1–6.