Automatic Tuning of Hyperparameters for Neural Networks in Serverless Cloud

Alex Kaplunovich
Department of Computer Science
University of Maryland
Baltimore, USA
akaplun1@umbc.edu

Yelena Yesha
Department of Computer Science
University of Maryland
Baltimore, USA
yeyesha@umbc.edu

2020 IEEE International Conference on Big Data (Big Data) | 978-1-7281-6251-5/20/$31.00 ©2020 IEEE | DOI: 10.1109/BigData50022.2020.9378280

***Abstract*—Deep Neural Networks are used to solve the most challenging world problems. In spite of the numerous advancements in the field, most of the models are being tuned manually. Experienced Data Scientists have to manually optimize hyperparameters, such as dropout rate, learning rate or number of neurons for Big Data applications. We have implemented a flexible automatic real-time hyperparameter tuning methodology. It works for arbitrary models written in Python and Keras. We also utilized state of the art Cloud services such as trigger based serverless computing (Lambda), and advanced GPU instances to implement automation, reliability and scalability.**

The existing tuning libraries, such as hyperopt, Scikit-Optimize or SageMaker, require developers to provide a list of hyperparameters and the range of their values manually. Our novel approach detects potential hyperparameters automatically from the source code, updates the original model to tune the parameters, runs the evaluation in the Cloud on spot instances, finds the optimal hyperparameters, and saves the results in the No-SQL database. The methodology can be applied to numerous Big Data Machine Learning systems.

Keywords—Neural Networks, Hyperparameter, Automation, Optimization, Big Data, Cloud, Serverless, Machine Learning, AWS

I. INTRODUCTION

Despite the popularity of neural networks, the step of hyperparameter tuning is manual for many data scientists. Although many researchers (including James Bergstra [1], [3], and [7]) were working on the problem, their solutions require identifying available hyperparameters manually before the tuning for each model (the hyperopt library[13]). The method of manual selection of the parameters is cumbersome, tedious and time consuming. Given the existing powerful parsers and available Cloud services, we have realized that it is possible to automate the whole process for an arbitrary neural network model.

Hyperparameters are the parameters needed to build a deep learning neural network model. They include:

- Number of layers
- Number of neurons
- Dropout rate
- Learning rate
- Number of epochs
- Activation function
- Optimization function

The first intuitive solution would be to run our model for all possible parameters combination (grid search). However, we should be aware of the curse of dimensionality [9] – the number of experiments grows exponentially with the number of parameters. For example, if we have 9 parameters, each taking 10 values, we will have to run our model 109 times to exhaustively search our dimension space. It is impractical, especially for large datasets.

James Bergstra et al. in [1] and [7] have demonstrated that randomly chosen parameters are much more efficient than exhaustive grid search or manual search. Moreover, since the number of parameters can be large, it will be physically impossible to try all the possible permutations to find the optimal set of hyperparameters. Our research is inspired by James Bergstra and his contribution to Machine Learning.

There are several hyperparameter optimization Python libraries available – Scikit-optimize [18], [19], and [21], RandomizedSearchCV [20] or hyperopt [13]. These libraries require a developer to create a dictionary of parameters and their values manually. We were unable to find an automatic tool that will be optimizing hyperparameters for an arbitrary model.

We decided to create a tool that will automatically optimize hyperparameters on assorted supervised neural networks (CNN, RNN, ANN, etc.). It is important that the whole process of the model hyperparameters detection and optimization is automatic. The input to our system is

1) Python/TensorFlow/Keras Machine Learning model code;

2) data files.

Once we received the input, we parse the model identifying hyperparameters to tune, apply dynamic code generation to make our model work with the hyperopt optimization library, and optimize our hyperparameters for the updated model code in the Cloud. The process is fully automated and triggered in the Cloud when a model is placed into an appropriate S3 bucket.

II. Proposed Approach

A. General

We will be providing the unified methodology to find optimal hyperparameters for an arbitrary Python ML model. We feel that it is necessary to design a tool that will pick the optimal hyperparameters for supervised deep learning programs, while allowing the data scientists to write their own code for data processing, model creation, and neural network business logic. We were unable to find such a system available. That is why we decided to create it.

Input to our system is a working Python machine learning model code along with a labeled dataset. The system will parse and analyze the code and find all the possible hyperparameters for tuning. Given the labeled dataset and the model code, our tool will find the optimal deep learning hyperparameters and output them. We implement the following steps:

- Keras layers hyperparameters spreadsheet creation;
- Parsing a Python machine learning model;
- Identifying the parameters;
- Updating the original model code;
- Running the updated model with the hyperopt library;
- Saving the results and sending notifications;
- Cloud integration.

B. Keras layers Metadata

To guide our system, we created an input csv spreadsheet containing hyperparameters metadata. It will be loaded by the system before it starts. Each line of the file corresponds to one hyperparameter and contains comprehensive information about it. We are planning to store the file in a shared location in the Cloud, to assure that all the instances of our system load the same parameter metadata. The file containing the following columns:

Layer, ParamName, Hptype, ParamOrder, Base, Type, Min, Max, Int, and Set.

These arguments help us to identify hyperparameters for Keras classes and layers. Below are the short descriptions of the columns:

Layer – a name of the Keras class representing a Neural Network layer (Dropout, Activation, Dense, etc.)

ParamName – a name of the hyperparameter (rate, activation, etc.)

Hptype – the type of the parameter values distribution from the hyperopt library (choice, uniform, quniform, etc.)

ParamOrder – order of the parameter in the class initialization list of arguments.

Base – base class name. Used if the same parameter is shared between a number of subclasses (for example, for multiple activations or optimizers)

Type – parameter type (n for number, s for string)

Min – minimum value for numeric parameters

Max – maximum value for numeric parameter

Int – a Boolean flag identifying that the parameter value should be integer

Set – set of parameter values for string parameters (for example, “relu, softmax, sigmoid” values for the activation parameter).

We would like to note that the file is internal, static and is loaded automatically by the system. The users do not have access to the files and should not change it.

C. Model Parsing

Source code parsing is a very powerful tool. Since it provides the access to the original code, it can figure out everything about the model and even transform if needed.

There are multiple parsing libraries available and we did our best to find the most reliable and flexible one, supporting indentation and Python3. We found that the AST event parser (using node visitor pattern) is the most appropriate for our tasks. It catches all the events of interest including assignment, expressions, method call, number, or print statement. The parser is so powerful that it can overwrite the node. During parsing, we

- collect all the variables and their assigned values;
- find all the DNN model layers and its parameters;
- replace hyperparameter values with the value placeholders compatible with the hyperopt library.

In a Python model, the same object can be created by multiple language constructs. Our parser is going to take care of all these cases including

- specifying the full or partial object class path
- object creation as a parameter
- object reference as a parameter
- implicit object creation assigned to a variable
- parameters passed by name or index
- functional model programming

```
model.add(keras.layers.Dense(128, activation='relu'))
model.add(Dense(128, activation='relu'))
model = Sequential([Dense(units=24,activation='relu'),...])
actv = keras.layers.ReLU(negative_slope=0.1)
model.add(Dense(128, activation= actv))
model.add(Dense(64, activation=ReLU(negative_slope=0.1)))
```

Fig. 1. Detecting assorted model constructed to detect hyperparameters.

Figure 1 demonstrates several use cases handled by our system and detecting the corresponding hyperparameters for future processing. Bold text in Figure 1 highlights the Keras classes and parameter values. For example, dense layer can be referenced as "keras.layers.Dense" or "Dense", and relu activation function can be passed as a string "relu" or as a variable "actv". We handle both sequential and functional programming.

D. Identifying the parameters

In this step, we create a parameter "space" used by the hyperopt library. The generated code includes parameter names and value ranges. The advantage of our approach is that we create the hyperparameter space argument automatically from the model source code. Our system automatically finds the hyperparameters from the model source code and generates all the necessary Python code to tune them calling hyperopt. We also insert all the required import statements into the generated code.

We are using the hyperopt library [13] for our tool. The library is the leading scientific approach [7] written by Bergsta et al. It provides powerful statistical methods [24] to tune the hyperparametrs using efficient random search algorithms (Bayesian optimization and Tree of Parzen Estimators). To identify the search space, we need to pass parameter "space" to the fmin method of hyperopt. Figure 2 demonstrates an example of generated space. The search space variable contains the dictionary of detected hyperparameters (to be tuned) along with their value ranges.

```
from hyperopt import fmin, tpe, hp
from keras import losses
space = {'window': hp.uniform('window',30, 100),
      'units1': hp.choice('units1', [64, 512]),
      'units2': hp.choice('units2', [64, 512]),
      'units3': hp.choice('units3', [64, 512]),
      'lr': hp.uniform('lr',0.01, 0.001, 0.0001),
      'activation': hp.choice('activation',['relu',
                                    'sigmoid',
                                    'tanh',
                                    'linear']),
      'loss': hp.choice('loss', [losses.logcosh,
                           losses.mse,
                           losses.mae,
                           losses.mape])
      }
```

Fig. 2. Generated space parameter for hyperopt.

The parameters can be of the following types:

- hp.choice – enumeration of the values;
- hp.uniform – specified every value in the range [min, max] equally likely;
- hp.quniform – uniform for integer values.

The hyperparameter reasonable values will be taken from the loaded csv file (described in Section II A). We would like to make our system flexible. A user, optionally, will be able to review the created parameter space and modify it if necessary. However, by default, our system will run automatically with the selected parameters.

E. Updating the original model

Once we have created the parameter "space", we can update our original model to run with the tuning library. Our parser allows us to see the offset of the Python statements. Our Machine Learning model can contain function definitions, imports or other statements. However, we can just create a method containing all the top-level statements. This method will be passed to the hyperopt's fmin method call to optimize the model. It definitely can call all the existing functions, making our approach flexible and efficient. The method will be creating our model with the parameters from the search space.

The AST parser is so convenient that it stores function definitions and blocks as one tree element with children. This structure helps us to group top-level statements together and place them into one function with domain space parameters. During this step we generate a code calling the hyperopt fmin function, including all the necessary imports and creating the actual model runner function passed to fmin. Figure 3 shows a code fragment generated to run the hyperopt tuning method fmin.

```
from hyperopt import fmin, tpe, hp, STATUS_OK, Trials
trials = Trials()
results = fmin(fn=experiment, space=space, algo=tpe.suggest,
max_evals=50, trials=trials)
```

Fig. 3. Generated Call to fmin – tuning method of hyperopt.

F. Editing the updated model (optional step)

There is an optional step, allowing users to edit the updated model before running the actual hyperparameter optimization step (II F). Data Scientist might want to look at the resulting code before running to verify it or to improve it. For example, we might want to move common code out of a method to call it just once. Loading data is a good example. If the generated code loads it for every iteration, we can move the code and load data just once. Given we might be dealing with Gigabytes of input files, it can decrease our code execution time drastically.

It can be possible to remove some of the detected hyperparameters if the author believes they should not be used

to improve her model or if the original parameter values are already optimal.

This step (if performed) can help us to eliminate model errors. Since the model author knows his code, they might want to perform additional improvements or optimizations before running the code in the Cloud.

G. *Running the updated model with hyperopt*

This is the most important step where we actually execute the model and tune its hyperparameters. All the preparations were targeted to run the fmin method of hyperopt. To recall, hyperopt [13] is a state of the art Python library for hyperparameter optimization written by Bergstra et al. Ironically, we are solving a dimensionality reduction problem.

We should note that running the model might take a lot of time even on the most powerful GPU instance. Another concern is cost. Although AWS Cloud provides very powerful machine learning GPU instances, they cost real money (even the discounted spot instances cost from $0.25/hour to $9.36 hour).

Fig. 3 demonstrates how to call the tuning methods using the space described earlier. The parameters include model creation method, search space, tuning algorithm, and number of evaluations. From a large space of potentially hundreds of hyperparameters (curse of dimensionality), we reduce our optimization problem by calling the method fmin with few parameters.

H. *Saving the results and sending notifications*

Upon completion, our system will save the detected optimal hyperparameters and model in a centralized location (S3 bucket and NoSQL DynamoDB database). It will send notifications via text message or email. Although it is possible to run the tuning step locally, it is better to run it on the cloud to save the results of our model optimizations at one centralized safe place and to provide almost infinite scalability. We might want to analyze the data in the future, visualize the results by assorted graphs and retrieve tuned models on demand. To send notifications, we use AWS Cloud SNS and SES services. The notifications will also be generated if an error occurs during the model optimization execution.

I. *Serverless Cloud integration*

Serverless – a new trend in Cloud Computing – is becoming mainstream. The granular code is deployed into the Cloud (as FAAS – Function as a Service), and the ecosystem takes care of container provision and termination, scaling and load balancing. By many industry experts, it is considered the best practice to utilize serverless architectures, if possible. Cloud function invocation can be automatically triggered by configured events, in our case a new model file saved into our S3 bucket.

Serverless is underrepresented in Machine Learning and our approach utilizes it. It helps to automate the process, improves scalability, resource utilization and assures zero idle server time.

As Peter Sbarski said in [27], “Serverless architectures are the latest advance for developers and organizations to think about, study, and adopt. This exciting new shift in architecture will grow quickly as software developers embrace compute services such as AWS Lambda. And, in many cases, serverless applications will be cheaper to run and faster to implement”.

Integration of our system with the Cloud ecosystem will solve several important issues:

- Results storage and aggregation – all results of our experiments will be stored at the same place, so we can incrementally analyze and graph them.
- Scalability – running multiple models in parallel on different machines.
- GPU Instance procurement – there is no need to have any local powerful computer. All instances will be created on the fly and terminated once our experiments are done.
- Automation – the optimization task will be triggered automatically once the model code is placed in the S3 cloud bucket; the results will be saved into the Cloud and a notification (text message or email) will be sent to the author upon completion of the model tuning.

These enhancements will help our system to be more robust, scalable, reliable and flexible. They will help us to run multiple models in parallel, potentially, with different datasets, providing unlimited number of powerful GPU instances. Figure 4 demonstrates the architecture of our platform. It is using the following AWS services – S3, AWS Lambda, DynamoDB, EC2, CloudWatch, SES, and SNS.

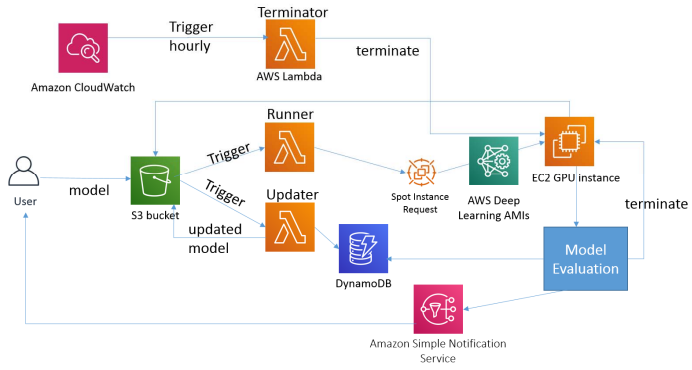

Fig. 4. Architectural diagram.

Our service is triggered by placing an original Python Model code into a certain storage directory, or an S3 bucket. We create a function trigger on S3 bucket(s) and a file object suffix (or prefix) to launch a function only when a model is placed into the bucket. We want our updater to start when a Python original model file (suffix Orig.py) is placed in the S3 cloud directory of our choice.

The Updater Lambda serverless function, updater, parses the model, extracts hyperparameters and dynamically creates the updated Model code that will be suitable to run hyperopt tuning. The updated model code is placed into the S3 bucket.

The function will parse the model and update its code for hyperparameter tuning as we have described in section 3.6. While the function is executed, the ecosystem will be monitoring it and saving logs to CloudWatch.

The second function, runner, is triggered when the updated model file is placed into the S3 bucket by the Updated function.

The Runner Serverless Lambda function spawns a GPU instance, runs the updated model and saves the results. It uses a spot instance template to create an EC2 instance with desired configuration. For the requested machine, we will be using one of the preconfigured Unix Deep Learning Machine Images (AMI, or Amazon Machine Image), containing most of the Python data science libraries. When we create an instance in the Cloud, we can pass numerous configuration parameters, including AMI ID. We can create our own AMI from an existing computer after installing and configuring all the necessary system components. For our instances we have been using Ubuntu operating system as one of the most reliable flavor of Unix. Upon completion, the runner attempts to terminate the instance.

Our third Lambda function, terminator, will be terminating instances. Although the updater function terminates its instance upon completion, we have created the terminator to catch and kill runaway instances. It is very important to be price cautious while using the Cloud. While we do our best to catch all the possible errors and failures during our system code execution, there is a slight possibility of something going wrong. In those cases, we lose control of the instance and cannot kill it. For such cases, we schedule the terminator to run hourly via CloudWatch event rules without any human interaction. The function, during its execution, checks for instances tagged “OptPar” specified during updater execution. If they are running for longer than 24 hours (a configurable parameter), the instances will be terminated.

Fast GPU computers are not always ready or available for Data Scientists. That is why Cloud integration helps us to spawn an instance as needed, and terminate it upon completion of our experiments. To save money on instances, we can even invoke spot instances using spot instance templates, saving up to 90% over on-demand price. The template helps us to automate the whole Cloud ecosystem infrastructure creation and orchestration with a single JSON configuration.

The code will execute hyperopt hyperparameter tuning, output results into the S3 bucket and notify the user via SNS service. Once the results are written, the third Lambda function will be triggered to terminate the created instance, to avoid any unnecessary charges and assure that we have zero idle time.

As a result, we will have all the model execution results in the S3 bucket for further analysis. Since Lambda invocations are asynchronous, our tool can handle multiple models at the same time. In our python code, we will have to access numerous AWS Cloud services. To accomplish it we will be using a boto3 library that has API access to almost every AWS cloud service.

III. Results

We were able to design methods to parse an arbitrary machine learning model and identify its hyperparameters. Our dynamic code generation was able to create a model that will be securely executed on the target GPU instance to tune the parameters for the requested search space.

TABLE I. Execution Results for most common models

Model	**Layers**	**Network Type/ #Params**	**Code Lines**	**Parsing (sec)**	**Execution Time (hour)**	**Loss/ Accuracy**	**Data Size**
Urban noises	5	DNN 8	200	5	1	0.02/ 0.89	4G
Bitcoin Forecast	4	DNN 8	190	5	50	0.1/ 0.87	.25G
Cat/dog	11	CNN 6	65	4	64	.003/ 0.95	.5G
MNIST	6	CNN 7	90	5	8	0.02/ 0.91	45MB
Translate	5	RNN 6	300	5	10	0.29/ 0.86	20 MB
Cifar100	7	CNN 28	100	5	6.15	1.2e-7/ 0.97	150 MB

We have run out tool on multiple models for multi gigabyte datasets; and the results were close to the optimal loss. Our method was able to parse the models successfully and identify the parameters automatically within seconds for diverse and complex models utilizing sequential and functional programming. Table 1 demonstrates the results of our experiments; we were able to achieve very small loss and high accuracy for the tuned models.

We have been covering most of the popular Big Data Machine Learning datasets (MNIST, cifar, cats/dogs) and

Deep Neural Networks layers – RNN with LSTM, CNN with Convolution, DNN with Dense, Activation, Optimizer, Dropout and many other classes provided by Keras. The diversified big datasets included images, audio, numeric and textual multiple columns data.

It is worth to mention that model tuning execution takes hours even on the most powerful multi-GPU instances AWS Cloud provides. That is why any optimizations we can do to the model, data loading or processing can be beneficial and can save time and money.

IV. Conclusion

Our approach improves and automates Machine Learning model hyperparameter optimization. While the existing state of the art methods require scientists to manually identify the parameters and choose their suggested ranges for optimization, our method takes the code and finds the parameters, suggests their ranges and runs all the necessary experiments in the cloud. We have verified that our methodology produces the same or better model loss and accuracy compared to existing hyperparameter optimization models.

We are planning to run our platform on a magnitude of models from Guthub and other sources. An innovative methodology detects and optimizes arbitrary Python models automatically. Our tool gives Data Scientists the power to create their own Python model using their experience, techniques and domain knowledge. Unlike several other machine learning automation tools, such as Google's AutoML or AWS's Machine Learning services, we do not oversimplify model creation and allow developers to write their own code. Cloud integration adds scalability, reliability and serverless integration to our system.

We are planning to improve our system to find the optimal tradeoff between cost, time and accuracy. Probably, we can run our tuning loads on subsets of data saving execution time and money. To accomplish that, we will need to find smart ways of splitting data reliably using the conclusions and findings from [26]. Because randomly generated standard splits do not always work, we will try to use multiple random splits.

Automatic hyperparameter tuning is vital for the whole Machine Learning community, and our methodology is definitely helping data scientists to optimize their models and improve application performance. During our Novel Coronavirus challenging times, we can tune models in all the research areas, especially those related to Big Data and Imaging. Nowadays, during the COVID-19 pandemic, well performing accurate models are especially important, because their predictions might affect billions of people.

References

[1] James Bergstra et.al. "Random Search for Hyper-Parameter Optimization", Journal of Ma-chine Learning Research 13, 2012

[2] Diederik P. Kingma, Jimmy Ba, "Adam: A Method for Stochastic Optimization"

[3] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, Balázs Kégl, "Algorithms for Hyper-Parameter Optimization"

[4] X.C.Guoab,J.H.Yanga,C.G.Wuac,C.Y.Wanga,Y.C.Lianga, "A novel LS-SVMs hyper-parameter selection based on particle swarm optimization"

[5] Nitish Srivastava et al., "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"

[6] G. Cybenko "Approximation by superpositions of a sigmoidal function"

[7] Bergstra, J., Yamins, D., Cox, D. D. (2013) Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. To appear in Proc. of the 30th International Conference on Machine Learning (ICML 2013)

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun,"Deep Residual Learning for Image Recognition", The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770-77

[9] Francis Bach "Breaking the Curse of Dimensionality with Convex Neural Networks", 4/2017, Journal of Machine Learning Research 18 (2017) 1-53

[10] About Keras models, Keras online documentation https://keras.io/models/about-keras-models/

[11] Metz, Cade "TensorFlow, Google's Open Source AI, Points to a Fast-Changing Hardware World", November, 2015

[12] Martín Abadi et al., "TensorFlow: A System for Large-Scale Machine Learning", November 2016, Proceedings of the 12th USENIX Symposium

[13] James Bergstra, Dan Yamins, David D. Cox "Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms", THE 12th PYTHON IN SCIENCE CONF, (SCIPY 2013)

[14] C.E. Rasmussen and C. Williams. Gaussian Processes for Machine Learning

[15] J. Mockus, V. Tiesis, and A. Zilinskas. The application of Bayesian methods for seeking the extremum. In L.C.W. Dixon and G.P. Szego, editors, Towards Global Optimization, volume 2, pages 117–129. North Holland, New York, 1978

[16] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In ICML 2007, pages 473–480, 2007

[17] C. Bishop. Neural networks for pattern recognition. 1995

[18] Fabian Pedregosa et al.,"Scikit-learn: Machine learning in Python", Journal of machine learning research, volume 12, October 2011

[19] "Tuning the hyper-parameters of an estimator", scikit-learn documentation, https://scikit-learn.org/stable/modules/grid_search.html

[20] RandomizedSearchCV documentation, https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html#sklearn.model_selection.RandomizedSearchCV

[21] skopt module documentation, https://scikit-optimize.github.io/

[22] Jonas Mockus: On Bayesian Methods for Seeking the Extremum. Optimization Techniques 1974: 400-404

[23] J. Mockus, V. Tiesis, and A. Zilinskas. "The application of Bayesian methods for seeking the extremum.", In L.C.W. Dixon and G.P. Szego, editors, Towards Global Optimization, volume 2, pages 117–129. North Holland, New York, 1978.

[24] Github hyperopt source and documentation, http://hyperopt.github.io/hyperopt/ and https://github.com/hyperopt/hyperopt/wiki/FMin

[25] C. E. Rasmussen and C. K. I. Williams. Gaussian Processes for Machine Learning. MIT Press, 2006.

[26] Kyle Gorman, Steven Bedrick, We need to talk about standard splits, Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL), August, 2019

[27] Peter Sbarski, "Serverless Architectures on AWS", 2017, Manning, ISBN 9781617293825