Hyperparameter adjustment in regression neural networks for predicting support case durations

Conference Paper *in* AIP Conference Proceedings · March 2021

DOI: 10.1063/5.0041936

2 authors:

Hristo Hristov
Free Varna University
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Galina Momcheva
Free Varna University
23 PUBLICATIONS **10** CITATIONS
SEE PROFILE

Hyperparameter adjustment in regression neural networks for predicting support case durations

Cite as: AIP Conference Proceedings **2333**, 070015 (2021); https://doi.org/10.1063/5.0041936
Published Online: 08 March 2021

Hristo Hristov, and Galina Momcheva

Hyperparameter Adjustment in Regression Neural Networks for Predicting Support Case Durations

Hristo Hristov [a] and Galina Momcheva [b]

Varna Free University, Yanko Slavchev 84, Chayka Resort, 9007 Varna, Bulgaria

[a] Corresponding author: hristo.hristov@vfu.bg
[b]galina.momcheva@vfu.bg

Abstract. Regression is a powerful technique for predicting a single scalar value with a high degree of certainty. A regression model requires a dataset that consists of only numeric features. However, datasets frequently contain both numeric and categorical features. This paper sets out to study several different text encoding techniques, which can solve this problem by transforming a certain text value into a corresponding numerical value that is statistically sane in relation to the dataset. The scenario we study is a neural network regression model predicting support case durations. Many of the dataset's features are indeed categorical such as issue description, team, username, severity and impact among others. The most challenging aspect of the encoding is the resulting change of the dimensionality of the dataset. Every encoding method affects the dimensionality in various degrees depending on the feature cardinality. This result is the main challenge for the tuning of the neural network hyperparameters. We must make such a setup that can robustly handle the altered dataset. The paper compares five different approaches for encoding: one-hot, hashing, binary, target and entity embeddings. The hyperparameter settings for each approach are presented by using common neural network performance metrics and a baseline neural network setup. We can conclude that a moderately increased dimensionality can enhance the model's predictive power as observed in the case of the binary and the hashing encoder

INTRODUCTION

Support systems have an integral role in the software development process. Once a product is released to production, an efficient and timely support process is critical to the product's overall success and perception among its target user base. In case of issues, changes or new requirements, users must be able to create support requests. These requests are then handled by operators that assign the cases to the relevant teams or individuals for resolution or feedback. The process of doing so is commonly referred to as a service model [1]. While working in the context of such a model, additional questions may arise for a specific case that sometimes can dramatically increase the handling time of each case or simply the cases require time and effort to debug. In many cases, additional information is required from vendors, off-site teams or other involved parties that could additionally prolong the resolution time. The impacted business users, in the meantime, are interested in their issue taking the lowest amount of time possible.

These factors allude that a support case is a highly non-linear function, i.e. it is very hard to make an educated guess of how long a certain case would take, based on which system is involved or which team is handling the case. For these reasons, we believe that a case duration prediction based on support system logs can provide substantial business value. It can approximate the time it would take to resolve a support case with a given degree of certainty. In this way, end users can have a virtual timeline and an expected deadline for the case resolution that does not rely only on hard-set SLA standards but rather on dynamic system information. By leveraging a regression neural network, the resolution time can be predicted for each case.

Applications of Mathematics in Engineering and Economics (AMEE'20)
AIP Conf. Proc. 2333, 070015-1–070015-9; https://doi.org/10.1063/5.0041936
Published by AIP Publishing. 978-0-7354-4077-7/$30.00

This work aims at exploring different methods for encoding categorical feature data that are used as inputs in said regression deep neural networks. This is an exercise in "manual" feature engineering as compared to the approach based on reinforcement learning presented by Zhang et al. [2]. In our case, we approach string encoding with deterministic methods that we study in detail.

Several different encoding methods are presented: one-hot encoding, target encoding, binary and hashing encoding. All these methods encode the source data in different ways resulting in more features than the model started with. The final method, entity embeddings is in a separate category, as it requires a different network architecture. The embeddings approach also manages to preserve inherent dependencies between categorical values, setting the ground for excellent data representation and generalization power [3]. We will, however, see that this approach is still not data agnostic. We have tried to preserve as much of the native continuous variables as possible in order to have a meaningful model [4]. However, the practitioner can rarely expect purely numerical features in a data set.

It is worth pointing out that all encoding methods depend on the feature cardinality. This is an important fact, especially for features of high cardinality. Such features pose a numerical problem [5]. Even more, as is the case with the one-hot encoder, they require more data points (samples) to work efficiently, i.e. the generalization power of the model becomes a partial function of certain features' cardinality.

All methods are applied on a business data set extracted from a support ticketing system. The dataset contains 141707 sample and 20 different features (columns) and has a native tabular format. Every row represents a support case instance. The features represent the various details about each instance, such as created date and time, support agent name, team, severity, impact, etc. The reader is encouraged to examine the dataset in [17]. For every encoding method, one regression neural network model is instantiated, trained and evaluated. Furthermore, we establish a baseline set of hyperparameters and a variable set of hyperparameters. The baseline parameters are those that do not change for all runs under the different encoding methods: activation functions, batch size, optimizer and regularizer values. The variable set of hyperparameters depends on the encoding method and their influence on the model is additionally studied. Method-specific hyperparameters are also subject to further optimization methods such as random search or Bayesian optimization [6]. In this work, however, we only go as far as explaining hyperparameters dependency for each model. The results are measured against the test dataset with the value of the mean squared error in order to determine which of the encoding methods can be ultimately recommended. Other performance metrics such as the generalization gap are also explored.

NEURAL NETWORK SETUP

Environment

For carrying out the training of the neural network, we use the free environment of Google Collaboratory in hardware accelerated mode. It offers up to 25.51 GB of RAM and 6-core Intel® Xeon® running at 2.30 GHz (3 physical cores with 2 virtual cores each). The computational load is executed on Tesla P100-PCIE-16GB supplied by Nvidia. This GPU is optimized for data center operations. The neural network training is environment-agnostic, however, if the reader wants to experiment, it is highly recommended to use a GPU for faster processing. While we work with a relatively small dataset with dimensionality of 141707 x 20 and up to 141707 x 8010, for encoding features in more complex datasets a more powerful environment may be required.

Hyperparameters

The following section describes the hyperparameter settings we used. The main assumption when setting the common hyperparameters was to have a sound baseline so the different encoding techniques could be compared. At the end of the section we list the method-specific hyperparameters. It is safe to say that they do depend on the dataset, so the reader must not take their settings for granted when experimenting with other datasets.

Activation Function

Two activation functions are used:

- ReLU, defined as $g(z) = max(0, z)$. This function is linear for inputs less than 0. This property makes it ideal for gradient-based optimization models [7].

- Linear, defined as $g(z) = g(z)$. If a unit does not transform its net input, it is said to have an "identity" or "linear" activation function.

Count of Neurons

For the hidden layers, these recommendations were considered:

- "A rule of thumb is for the size of this [hidden] layer to be somewhere between the input layer size and the output layer size" [8].
- it should never be more than twice as large as the input layer [9].

After an analysis on these relevant facts and after several test runs, the following counts are selected, Table 1:

TABLE 1. Count of neurons

	Hidden Layer 1	**Hidden Layer 2**
One-hot encoder	40	20
Target, binary, hashing encoder	20	10
Entity embeddings	1000	512

We are attempting to balance the increased dimensionality of the one-hot encoder with more neurons in the hidden layer.

Count of Layers

Because the dataset in this case is relatively simple (e.g. not related to computer vision), we have chosen to use just two hidden layers. The assumption is that two layers can approximate the target value with relatively acceptable loss. This assumption was successfully justified to the degree of the test loss as recorded for each model.

Optimizer

For our setup we used the Adam optimizer [10]. The default parameters follow those provided in the original research paper: keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=**False**)

Batch Size

The batch size b is always $1 < b < n$**,** where **n** is the number of samples in the dataset. If b is closer to 1, that means very few test cases are being processed per each iteration of the network.

In our case we have chosen to set the batch size to 512. This number is chosen with regard to the number of cases in the training data which is $0.6 \times 141\,707 = 85\,024$, although there is no set rule that establish a relation between how big the training set is relative to the batch size. For our purposes, we get $85\,024 \div 512 \cong 166$ parameter updates per epoch.

Epoch Count

The number of epochs is a hyperparameter of gradient descent which sets the number of complete passes through the entire training dataset.

In a way, the epoch count is a temporal concept. Given enough time, the network will learn the best approximation. However, given infinite time, i.e. if epochs $\mathbf{e} \to \infty$**,** a good predictive capacity is not always guaranteed. The reason is that it is hardly the only influencing factor as seen already. Therefore, the number of epochs must be optimally chosen so that it allows efficient training for the given parameter count over the training dataset. In our scenario, we perform two training rounds. The first one is 10 epochs long. Such a short training time allows us to zoom in on the model behavior very early in the training process. The second training round is carried out over the course of 100 epochs. This longer training time allows the model to make more adjustments to its parameters and provide a better overview of the learning process.

Loss Function

The loss function for regression models, by convention, is the mean squared error (MSE). The MSE is a measure according to the formula [14]:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{f}(x_i)\right)^2 \qquad (1)$$

where $\hat{f}(x_i)$ is the prediction that the objective function $\hat{f}$ provides for the *i*th observation. We provide the MSE values for all train, validation and test datasets.

Regularizers

The regularizer applied is only L2 ("weight decay") [11] with a standard value of 0.001. Additional regularization was considered but decided not to be applied. Our motivation for this choice is that we want to establish a reasonable common baseline where the best categorical encoder can naturally stand out. The assumption is that with additional regularization applied, the best model could be further optimized.

Output Unit Form

Since we are working on a regression scenario, the output unit's shape is 1. That means there is one scalar value that we train the network to predict – the support case duration value.

Method-specific Hyperparameters

- Target encoding weight:

The target encoding weight is the first method-specific hyperparameter. It is the number m that we assign to the overall mean. Generally, this number is determining how many values from the sample mean are required to overtake the global mean. The initial value is $m = 5$. We make an experimental run with $m = 100$ as well.

- Hashing encoding vector size:

The hashing encoding approach involves another extra parameter n referred to as the number of components or vector output size[12]. We perform two runs with $n = 15$ and $n = 50$.

- Embedding size:

The methodology of the entity embedding requires specifying the entity dimension e. This is variable-size vector, with a maximum value $e = 50$. The actual value is set by this formula:

$$e = int(min(np.ceil((no\_of\_unique\_cat) / 2), 50)$$

An extra model run is performed with a size $e = 100$ as well to measure the differences between different embedding sizes. Each categorical feature value will be represented by values in this vector.

Metrics

For measuring the performance of each model, we consider the following three metrics:

1. The training and validation learning curves. Both are a representation of the error (y-axis) over the temporal dimension (epochs: x-axis). They reveal how representative the data set is and how the model can score overall:
    - Training dataset loss curve represents how well the model is learning.

- Validation dataset loss curve represents how well the model is generalizing.

2. The performance of the model on the test set L_{test}. The evaluation is based on the final mean squared error value over the test dataset. The lower the value, the better.
3. We also consider the difference in the generalization gaps between the two runs of each network. The first run is 10 epochs long, while the second run is 100 epochs long.

This measure *ΔGl* is expressed in the following way:

$$\Delta G_l = G_{l100} - G_{l10} \qquad (2)$$

It indicates if the generalization gap is decreasing or increasing as the training proceeds. Ideally, we are looking for negative or values of $\Delta G_l \rightarrow 0$ indicating a convergence in the generalization gap.

Ultimately, we consider the generalization gap as an indicator of how good a model is. It is defined as the difference between the loss value from the training dataset and the loss value of the validation dataset [13]. In an ideal scenario, this difference needs to be very small, while the training loss should remain higher than the validation loss throughout the training period.

We propose a measure, G_l (Generalization gap equation) that measures the difference between the validation mean squared valued (MSE) and the training MSE:

$$G_l = MSE_{Validation} - MSE_{Training} \qquad (3)$$

We have found the following to be true according to the result of equation (3):

- $G_l > 0$ means the validation error is greater than the training error. This is typically the expected result, as the model tries to predict the output based on unknown inputs from the validation dataset. Naturally these values are harder to predict than the training dataset.
- $G_l < 0$ suggest that the validation set has been easier to predict. This scenario does occur sometimes under some encoding methods. The observation does not inherently mean a disadvantage of the model; however, it requires further investigation of the dataset because the assumption is that the validation data has been easier to predict than the training data.

For the final scoring, we consider only the positive values. The measure does not have bounds, but we are looking for values of G_l that are as small as possible in the given context.

The generalization gap itself is a metric that can characterize a model at a glance:

- If the validation loss curve is higher that the training loss curve, we could have a training dataset that is not enough for the model to learn from; hence the model cannot generalize well.
- If the training loss curve is higher than the validation loss curve, it means that the model is better at predicting the unseen values rather than the "seen" values of the training dataset. Such a scenario may indicate a problem with the validation set or the fact that the encoded data has poor representation of the original data.

In an ideal setting, we observe convergent training and validation loss curves; even more, the two curves level off and can appear parallel with some inherent fluctuations. The gap between them should be as small as possible, although there is no fixed value indicating what a good gap is.

For each of our model we present the plots that clearly show the curves and the generalization gaps. In this way, the better models can be easily identified and studied further. The lesser models can also be studied further by making them a subject of subsequent optimization, hyperparameter tuning or enhanced feature engineering.

COMPARATIVE ANALYSIS

Starting with the one-hot encoding approach, we see the very low values of the training loss and the substantially higher values of the validation loss. As observed in the plot, this is already an indication of overfitting and poor performance. The fact is confirmed by the high values of the test loss which makes the model impossible to use in

practice for getting a good prediction. The generalization gap is consistently higher and diverging – it grows from 0.7777 to 0.8741.

The target encoding method results deliver some promising figures. The training and validation loss values are close to each other for both sets of runs with a weight of 5 and 100 respectively. With w=100 we observe more converging training and validation loss curves, however, w =5 yields a better result with a test loss of 0.5618. As we will see in the next paragraphs, this is the best performing model.

After implementing the binary encoder, we observe results that stand with the moderately increased dimensionality **d** from 20 to 121. Training loss is decreasing with time from 0.5168 to 0.2365, however the final test loss after 100 epochs is over 1.5. This model could be useful if trained for shorter periods of time. The training and validation losses are also divergent as indicated by $\Delta Gl = 0.4643$.

Next we examine the hashing encoding method. We have a total of 4 sets of runs depending on the dimensionality, hashed values standardization and epoch count. The models with default values and dimensionality of 20 performs best, ending up with test loss of 0.7306 and 0.7331 respectively. Overall, the hashing encoder is the second-best scoring method after the target encoder. This is an important conclusion that proves that hashed index value representation is also an efficient statistical approximation, as compared to the smooth mean of the target encoder. This approach also exhibits efficient running times. If we had to extrapolate those running times for future epochs greater than 1000 and big data, the hashing encoder could be a recommended approach.

We finally observe the metrics for the entity embeddings approach. We have two sets of models, depending on the vector sizes and the epoch counts. All models exhibit low training losses, less than 0.04 and high validation loss resulting in high *Gl* values of over 0.75. All resulting test loss values are over 1.0 suggesting inapplicable performance in real-life scenarios. Despite that, the embedding of size 100 being run for 10 epochs shows the lowest test loss. Overall, this encoding approach is the third worst performing one, after the binary and the one-hot encoding methods.

In summary, we can conclude that a moderately increased dimensionality can enhance the model's predictive power as observed in the case of the binary and the hashing encoder. On the other hand, a drastically increased dimensionality is not a guarantee of good predictive powers. The performance of the other methods over different datasets with increased dimensionality can be examined in future work.

The plot in Table 1 shows all model metrics laid out in horizontal bars. The metrics are ordered first by test lost, then by training loss in an ascending order each.

TABLE 2. Model metrics

Encoding method	Epochs	d	$Loss_{train}$	$Loss_{val}$	$Loss_{test}$	Gl	ΔGl
one-hot I	10	8010	0.0177	0.7949	1.0902	0.7777	
one-hot II	100	8010	0.0042	0.8776	1.2711	0.8741	0.0964
target I w5	10	20	0.3865	0.4237	0.5366	0.0464	
target II w5	100	20	0.2962	0.4772	0.5618	0.1839	0.1375
target I w100	10	20	0.4363	0.4520	0.6064	0.0225	
target II w100	100	20	0.3390	0.4236	0.6708	0.0874	0.0649
binary I	10	121	0.5168	0.7240	0.7429	0.2096	
binary II	100	121	0.2635	0.9362	1.6971	0.6739	0.4643
hashing I - default	10	20	0.9817	0.8485	0.7306	-0.1268	
hashing II - default	100	20	0.8860	0.8525	0.7331	-0.0315	0.0953
hashing III - default	10	55	0.6940	0.7553	0.7360	0.0663	
hashing IV - default	100	55	0.4882	0.8556	0.8511	0.3685	0.3022
hashing I - normalized	10	20	0.9985	0.8753	0.7538	-0.1165	
hashing II - normalized	100	20	0.9178	0.8643	0.7419	-0.0516	0.0649
hashing III - normalized	10	55	0.6920	0.7426	0.7642	0.0545	

TABLE 2 (Continued). Model metrics

Encoding method	Epochs	d	$Loss_{train}$	$Loss_{val}$	$Loss_{test}$	Gl	ΔGl
hashing IV - normalized	100	55	0.4633	0.8459	0.8517	0.3834	0.3289
embeddings - v50 I	10	20	0.0356	0.8856	1.1423	0.8500	
embeddings - v50 II	100	20	0.0103	0.9805	1.2280	0.9702	0.1202
embeddings – v100 I	10	20	0.0334	0.8171	1.070	0.7837	
embeddings – v100 II	100	20	0.0108	0.8828	1.1383	0.8720	0.0883

Performance visualization

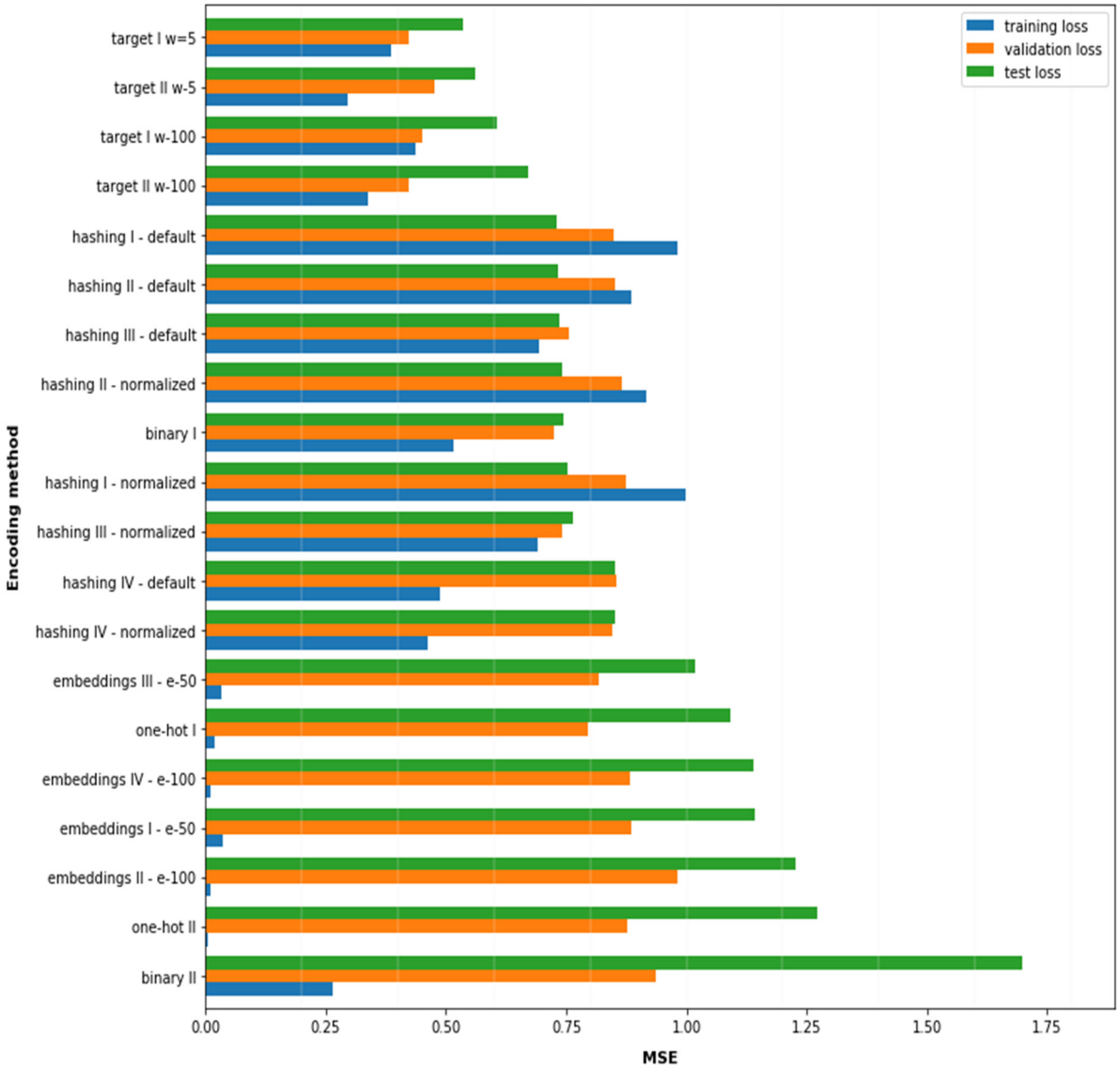

FIGURE 1. Performance visualization for all models

CONCLUSION

With this work we hope to have established an initial overview of a selected few categorical encoding methods. The original paper is by no means exhaustive; many other methods exist that have not been examined here. Some of them are: simple coding, deviation coding, difference coding, Helmert coding, orthogonal polynomial coding, repeated coding [15], among others. This multitude of available methods provides the luxury of choice, but it does not directly help a given encoding task. As we have seen so far, the performance of each methods is a function of the data. More specifically, it is a function of the statistical representation of the data that the method produces when converting the string values to numerical values.

Categorical encoding methods are not relevant to regressions problems only. However, the starting point for each regression problem is the tabular dataset. Therefore, datasets can be found in many different forms across different business systems. It could be an excel file with supplier performance for the past year or a huge database with accounting information. Whatever the data or context is, any tabular dataset can be engineered to be the input for a neural network. This effort can provide enormous value in day-to-day business operations.

With that in mind, it is important to also mention the integration options for neural networks. Content is rapidly created, and data is everywhere so why should not be a neural network available to provide insight on that data? Categorical encoding can ultimately help remove certain challenges in such a scenario. For example, tools from various software vendors exist already for embedding neural networks into database systems and collaboration platforms [16]. Even more, neural network modelling and advanced analytics should be considered at the start of any systems design and architecture.

We hope to have contributed to future efforts of making neural network an inherent part of databases and business systems.

ACKNOWLEDGMENTS

The results analyzed in this article have been obtained during the writing of a master's thesis for a M. Sc. in Data Science. Its full text and accompanying code samples can be found at the link in [17]. The BioMed Varna academic workgroup, to which both authors belong, will further use these research findings for its future work. Sincere thanks go to our academic supervisors assoc. prof. Galina Momcheva, assoc. prof. Eddy Chakarov and the editors of this article for their continued support, guidance and academic advice.

REFERENCES

1. T. Nechyporenko, ServiceNow As A Platform – Practical Research, Bachelor's Thesis, Haaga-Helia University of Applied Sciences, Finland, 2015
2. J. Zhang, et al., Automatic Feature Engineering by Deep Reinforcement Learning, College of Intelligence and Computing, Tianjin University (May 2019)
3. C. Guo, F. Berkhan, Entity Embeddings of Categorical Values (May 2016)
4. F. E. Harrell, Regression Modelling Strategies (Springer International Publishing, Switzerland, 2015), p. 8.
5. M. Kuhn and K. Johnsson, Feature Engineering and Selection, (CRC Press, Boca Raton, FL, 2020), p.96
6. J.N. van Rijn, F. Hutter, Hyperparameter Importance Across Datasets, Albert-Ludwigs-Universität Freiburg, 2018
7. I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, (MIT Press, 2016), p.189
8. A. Blum, Neural Networks in C++, (Wiley, New York, 1992)
9. M.J.A. Berry, and G. Linoff, Data Mining Techniques, (John Wiley & Sons, New York, 1997)
10. D.P. Kingma, and J. Ba, Adam: A Method for Stochastic Optimization
11. L. N. Smith, A Disciplined Approach to Neural Network Hyperparameters: Part I, US Naval Research Labor, (04.2018)
12. K. Weinberger, Feature Hashing for Large Scale Multitask Learning (Yahoo Research, Santa Clara CA, 2014)
13. Y. Jiang, D. Krishnan, H. Mobahi, S. Bengio, Predicting the Generalization Gap in Deep Networks with Margin Distributions, (Google Research, conference paper at ICLR, 2019)
14. G. James, D. Witten, T. Hastie and R. Tibshirani, An Introduction to Statistical Learning (Springer, New York, 2017)

15. Coding Systems for Categorical Variables In Regression Analysis. UCLA: Statistical Consulting Group. from https://stats.idre.ucla.edu/sas/modules/sas-learning-moduleintroduction-to-the-features-of-sas/ (accessed May 14, 2020)
16. E. Schikuta, Neural Networks and Database Systems (University of Vienna, February 2008)
17. H. Hristov, "Hyperparameter Adjustment in Regression-Based Neural Networks for Predicting Support Case Durations" (Varna Free University 2020) from https://github.com/hristochr/thesis (accessed May 10, 2020)