Варненски Свободен Университет „Черноризец Храбър“

Факултет „Международна икономика и администрация“

Катедра „Информатика и икономика“

Master's Thesis

Hyperparameter Adjustment in Regression-based Neural Networks for Predicting Support Case Durations

~

Настройване на хиперпараметри при невронни мрежи базирани на данни от съпорт системи

Hristo Hristov

Data Science

Student ID: 182831001

Research supervisor: Assoc. Prof. Galina Momcheva

Varna, Bulgaria, IV. 2020

Contents

1. Introduction

1.1. Aims, research tasks and problem overview

This work aims at exploring different methods for encoding categorical data used in regression-based deep neural networks. Several different encoding methods are studied here: one-hot encoding, target encoding, binary and hashing encoding. The final method, entity embeddings is in a group of its own as it requires a different network topology.

All methods are explained and applied on a business data set extracted from a support ticketing system. For every encoding method, one 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. Hyperparameters dependency for each model is explained. 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.

1.1.1. What is a prediction?

In principle, neural networks can compute any computable function, i.e., they can do everything a normal digital computer can do [Valiant, 1988; Siegelmann and Sontag, 1999; Orponen, 2000; Sima and Orponen, 2001] under some assumptions of doubtful practicality [Siegelmann, 1998; Hadley, 1999].

Practical applications of neural networks often involve supervised learning. In supervised learning, as opposed to unsupervised learning, the data engineer must provide a training data set that includes both the input and the desired result, also referred to as the regressand or the target value. After a successful training episode, you can present input data alone to the neural network (that is, input data without the desired result), and it will compute an output value that approximates the target. However, for training to be successful, the engineer must provide lots of training data and lots of computing resources and time to do an effective training. In many applications, such as image and text processing, engineers must do a lot of pre-processing and data transformation to select appropriate input data and to encode the data as numeric values. This type of supervised learning with quantitative target values explored here is called "regression."

1.1.2. Business case

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 or doubts, 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. In the process of doing so, additional questions may arise 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 sub-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 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 the regression model and deep learning, the resolution time can be predicted for each case.

1.1.3. Why an artificial neural network?

Neural networks are an incarnation of deep learning that do powerful supervised learning. In theory, by adding more layers and more units within each layer, a deep network could approximate functions of varying and increasing complexity. Most scenarios that involve mapping an input vector to an output vector could be implemented via deep learning. The prerequisite is to have a proper model and enough training data so that the model can learn from it. [Goodfellow , A.Deep Learning, 2016]

We have these conditions fulfilled: data for more than 20 000 support cases are available and two types of models for handling it. Additionally, the neural network produces specific output in terms of the test error. Therefore, the different models can be compared efficiently. Thus, we comply with the definition “Artificial neural systems, or neural networks, are physical cellular systems which can acquire, store, and utilize experiential knowledge.” [Zurada, J.M. Introduction To Artificial Neural Systems]

1.2. Process mining aspect on the data

This dataset is pertaining to a specific business flow – the support process in enterprises. Therefore, it is worth mentioning the applicability of process mining techniques to the data as it can provide additional insight.

Process mining is noteworthy for these reasons:

- Prediction based on process data. This is our focus throughout the work from the regression standpoint
- Feature utilization – in process mining we can leverage features that are not necessarily part of the input vectors thus exploiting the data to the

fullest. These features are **number**, **incident_state**, **sys_updated_at** which we drop out of the input variables as described in 1.3.1.

These features, however, are vital to the process mining approach making it complementary to the neural network prediction. We particularly need these features to comply with the typical event log which must consist of a case identifier, event identifier and timestamp. [van der Aalst, W. 2012. Process mining: Overview and opportunities. ACM Trans. Manage].

Original attribute	Definition naming
Number	Case ID
Incident state	Event ID
Updated at	timestamp

Table 1. Attribute mapping to event log definition

This attribute mapping of the original features to the relevant log definitions will allow to employ Pm4Py[1] for generating a heuristics net out of the complete dataset. The heuristics net provides visual information on how each case is progressing and on how the different events are interconnected:

[1] http://pm4py.org/

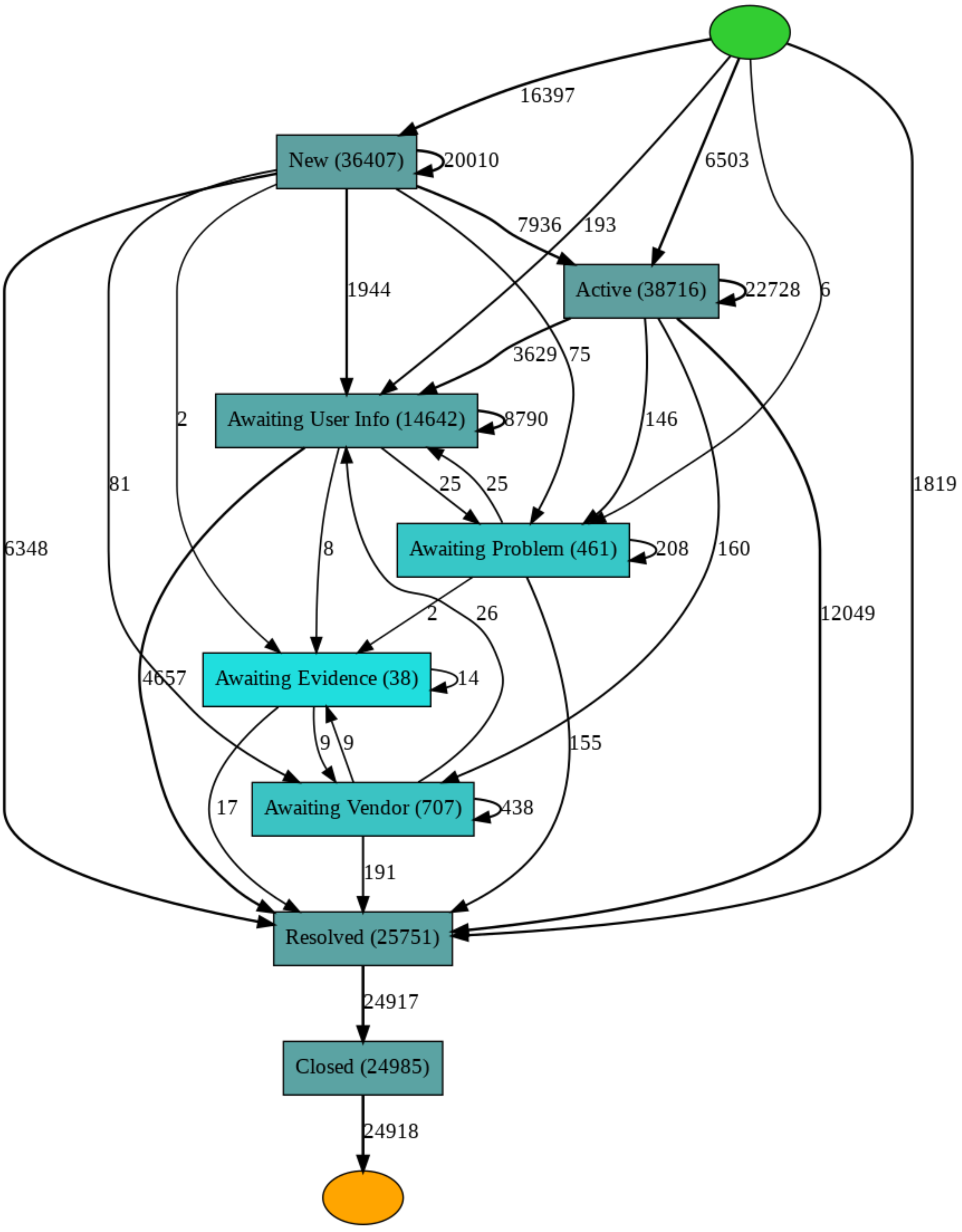


Figure 1. Heuristics net

1.3. Features

The format of the data we work with is tabular. It provides a clear structure with valuable knowledge. [Deng, L., Zhang, S., Balog, K., Table2Vec: NeuralWord and Entity Embeddings for Table Population and Retrieval]. Having the data in tabular form is a hard prerequisite for performing regression analysis and predictions. The inputs for the neural network are the independent values, i.e. the values we have as given. We try to extract the relevant information from them in order to provide a prediction for the dependent value. In our case that value is the case duration.

1.3.1. Cursory data description

The raw dataset consists of the following feature vectors:

Name	Description	Example	Dropped
number	incident number	INC0000045	yes
incident_state	Current status of the incident	New	yes
active	Denotes if the incident is active or not	true	yes
reassignment_count	Number of times the incident has been reassigned	1	No
reopen_count	Number of times the incident has been reopened after being closed/resolved	0	no
sys_mod_count	System modification count	0	yes
made_sla	Denotes if the incident complied with the target SLA	true	No
caller_id	Who created the incident, anonymized	Caller 240	no
opened_by	Who opened the ticket, anonymized	Opened by 8	yes
opened_at	Time and date the ticket was opened	29/02/2016 01:16:00	no
sys_created_by	System value	Created by 6	yes
sys_created_at	System value	29/02/2016 01:23:00	yes
sys_updated_by	System value	Updated by 21	no
sys_updated_at	System value	29/02/2016 01:23:00	yes
contact_type	How was the incident raised	Email	
location	Location information about the incident, anonymized	Location 143	No
category	Incident category, anonymized	Category 55	No

subcategory	Incident subcategory, anonymized	Subcategory 170	no
u_symptom	Information about incident symptom	Symptom 72	No
cmdb_ci	Application identificatory	-	no
impact	How impactful on operations the issue is	2-Medium	no
urgency	How urgent the issue is	3 - Low	no
Priority	Incident priority relative to other incidents	3 - Moderate	no
assignment_group	Support group, anonymized	Group 56	no
assigned_to	Currently assigned support engineer, anonymized	Resolver 31	no
knowledge	Denotes if a knowledge article is available	TRUE	yes
u_priority_confirmation	I	FALSE	yes
notify		Do Not Notify	yes
problem_id	Problem identification	Problem ID 2	no
rfc		CHG0001526	yes
vendor	Vendor name, anonymized	Vendor 1	no
caused_by	Information on incident cause	CHG0000097	yes
closed_code	Incident outcome	code 5	yes
resolved_by	Who resolved the issue	Resolved by 149	no
resolved_at	Date and time the issue was resolved	01/03/2016 09:52	yes
closed_at	Date and time the issue was closed	06/03/2016 10:00	no

Table 2. Overview of the features. Several were dropped from the start due to missing values or because they were considered to have low importance to the model

In Table 2. we can see all features available in the data set. After the drop, we have 20 input features to work with. There is no specific methodology employed regarding which features to drop. The decision was taken based on determining how applicable the specific feature is to the prediction. For example, we have determined that the **incident_state** feature should not be part of the inputs as we don’t want the current state to influence the outcome for the prediction.

1.3.2. Quantitative feature analysis

In this section we expose some of the basic characteristics of some of the numerical features in the dataset. By doing so, we want to get a deeper look into the data. In a real-world scenario, such an analysis would be the baseline starting point for further examination. Here we would like to get a broad idea of how non-linear the dependency of case duration is on the input features. We performed the analysis by using the seaborn Python library for statistical visualizations.

1.3.2.1. Opened at histogram

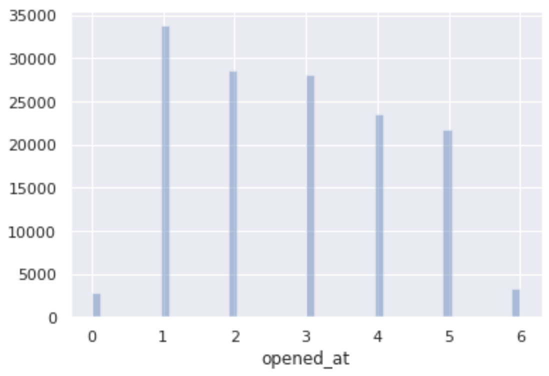


Figure 2. "Opened at" histogram

Fig. 2. shows the histogram of the values in the "***opened at***" feature vector. The original values in the data set were of type date and time in the format dd/mm/yyyy hh:mm. For our purposes, we have considered some of this data and time information redundant. Therefore, the values have been converted to show only the day of the week. In our context this bears much more information and could prove to be a strong influencer on the case duration.

From the conversion we get numbers in range 0 to 6. The number zero represents Sunday, while six is Saturday. On the plot we observe that very few – less than 5000 cases are opened in the weekends. Most of the cases – about 34 000 occur on Mondays, with the rest of the business days stay in the range of about 22 000 – 29 000 occurrences per day. More importantly, converting the original date and time representation to integer representation of weekdays, we get a straight numerical representation of the dates. Another approach would be to convert the dates to a Unix timestamp.[2]

1.3.2.2. Case duration distribution

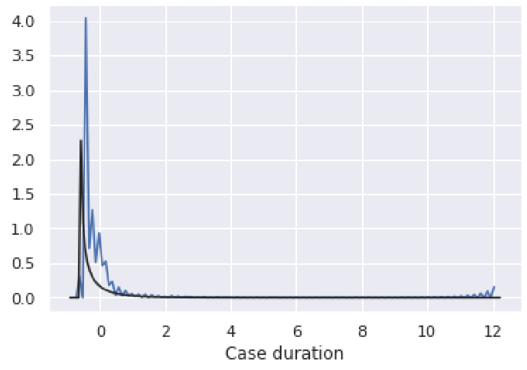


Figure 3. Kernel density estimation for "case duration".

The case duration values are calculated by subtracting the “***opened at***” values from the “***closed at***” values for each incident case. The result is a floating-point number representing the number of days it took to resolve that specific case. Ultimately this is also the value we want to predict for future case instances. We refer to it from here on as the target value, or dependent value. In a real-world implementation, business users would be interested in having an automatic estimation from the support system on how long their case could take. The categorical encoding methods aim at providing this estimation with a high degree of certainty, as suggested by the test loss. The values are presented for each method accordingly.

On Fig. 3., we observe the fact that a great amount of the cases appears to take less than a day to resolve. It is also safe to state that most of the cases take less than 2 days to resolve. The fitted black curve is the gamma distribution, showing that that the KDE of the case duration follows it closely.

[2] Unix timestamp conversion: https://www.epochconverter.com/

1.3.2.3. Numerical feature values - pairplot analysis

By using the seaborn pairplot() method we can get a pairwise plot showing the bivariate distributions in the dataset. The resulting square matrix *P* of dimension 6 x 6 shows the relationship between each pair of columns. In the diagonal, the univariate distributions of each feature are observed.

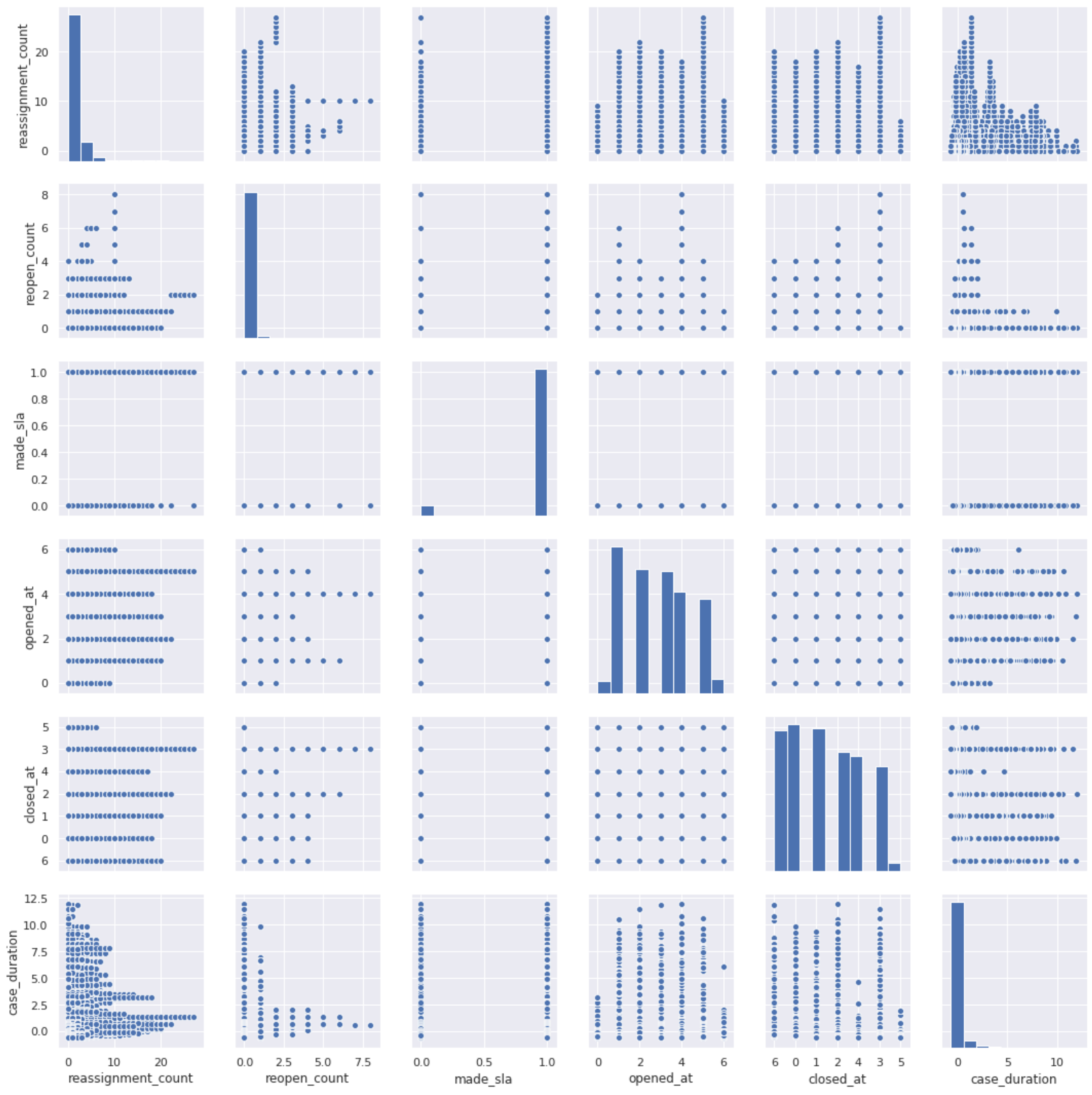


Figure 4. Pairwise feature analysis, matrix P

At position (6,1) observable is the fact that the longer a case takes, the less reassignments it tends to have. At position (1,2) we can also see that more reassignments tend to happen for cases that were reopened less than 5 times. At position (4,4) we see the familiar histogram of the values of the “opened at” feature converted to weekday numerical representation. At position (4,6) the correlation of opened at and case duration clearly shows that cases opened on Saturdays and Sundays tend to require less time to resolve. On the other hand, Wednesdays and Thursdays tend to produce the cases that take longer to resolve.

1.3.3. Missing values

We have determined to eliminate the cases that happen to contain missing values for any feature. On the one hand, the missing values can greatly affect the model’s predictive power. On the other hand, we do not need to make predictions based an input vector with missing values.

We use the *na_values* parameter of the *read_csv* method in pandas by providing the values “NA” and “?”. The dataset examination showed that these are the only values that are in fact representing missing values. Those are substituted with a constant value “No {feature_name_placeholder}”. This ensures consistent and relevant treatment of the missing values.

1.3.4. Features categorization

For the purpose of this thesis, we divide the features into two broad categories: numerical and categorical features. A brief definition is provided here.

1.3.4.1. Numerical

Numerical features are represented by numbers: either real numbers or integer numbers. Our dataset does not possess any features with real numbers values natively. Only the dependent feature – case duration results in a real value after being calculated from the opened at and closed at time and date values.

The integer-valued features we have are the reassignment count, reopen count, made SLA (1 or 0), opened at and closed at (parsed into a weekday representation).

1.3.4.2. Categorical

The second type of features found in practice in tabular datasets is the categorical type. These are string values which are typical metadata in a business context or when data is extracted from business systems. The challenge is not so much to analyse, but rather to represent those string values in a statistically suitable way for the model to be able to interpret them. As we will see, interpretability greatly depends on the statistical representation. Different approaches in categorical features encoding can lead to vastly different results. We use our baseline neural network architecture and the metrics established to evaluate all approaches accordingly.

For example, the first approach presented is the one-hot encoding. For high cardinality categories, it is a method proven to lead to high-dimensionality feature vectors. [Cerda, P., Encoding high-cardinality string categorical variables]. In turn, this fact leads to computational challenges, such as processing thousands of features, but also data representation problems due to sparsity (the model is having a hard time extracting the relevant information based on the input vector as it contains mainly zeros). The former fact may require additional work on the dataset, such as some of dimensionality reduction.

We are going to use the following notations from here on for describing the dataset:

Symbol	**Definition**
$\boldsymbol{X}$	Matrix representation of the dataset
$\boldsymbol{x_i}$	Row vector or element of the vector space X
$\boldsymbol{S}$	Set of all finite-length strings

$\boldsymbol{C}$	Categorical variable or column vector
$\boldsymbol{n}$	Number of samples, interchangeably referred to as cases or records
$\boldsymbol{d}$	Dimensionality of the categorical encoder
$\boldsymbol{y}$	Target variable (case duration)

Table 3. Notations used throughout the work

In Table 3. we show the notations we are going to use in the thesis. They are subject to the following rules by definition:

- $i = 1 \dots n$
- $n = 141\,707$**,** which is the overall number of rows in our dataset
- $x \leq X$**,** meaning each row vector is a subspace of the dataset
- $C \leq X$**,** meaning each column vector is a subspace of the dataset
- $C \subseteq S$, all column vectors are a subset of all finite-length strings
- $d_s = 20$**,** i.e. the starting dimensionality is 20 input vectors
- $d_y = 1$**,** i.e. the dimensionality of the target variable is 1, because this is a regression problem.

2. ANN setup and specification

2.1. Physical environment

For the experimental part of this thesis, 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.

Neural networks frequently require a lot of computational power especially when there is a lot of data to process. Training such networks on personal hardware is often inefficient.

2.2. Hyperparameter setup

The main challenge when creating a deep learning model is the hyperparameter selection. The hyperparameters are the values that effectively define the model. They can be considered as multiple knobs and switches that must be adjusted prior to instantiating and running the network. We have listed those in the points below.

The challenge is that hyperparameters in practice belong to different domains. [Diza, G., An effective algorithm for hyperparameter optimization of neural networks] They could:

- have varying upper and lower bounds, e.g. layer count and neuron count;
- be positive or negative;
- be integer or floating-point numbers, e.g. epoch count and regularizer values;
- be categorical, such as the name of the optimizer function.

For the setup in this thesis, we have followed several good practices explained per each hyperparameter next. Many hyperparameters related to the method of learning utilized – stochastic gradient descent, are in fact omitted. The reason for the omission is that we do not need to configure them explicitly when using Keras. Default values supplied by the Keras library are used.

2.2.1. Activation function

The units (neurons) in neural networks transform their net input by using a scalar-to-scalar function referred to as an "activation function". It returns a value called the unit's activation. Except for the output unit (in our case just one such unit), the returned activation value is passed via synaptic connections to one or more other units. The activation function is sometimes called a "transfer", and activation functions with a bounded range are often called "squashing" functions. For our model 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 [Goodfellow , A.Deep Learning, 2016]
- 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.

The linear function is used only for transferring the input form the second hidden layer to the output layer. In a regression model, squashing the input between these two layers is not required, nor does it bring extra value for the predictive powers of the model.

2.2.2. Count of neurons

The dimensionality of the input data is 141707 x 20, meaning 141707 records and 20 independent features. Therefore, the count of neurons in the input layer is chosen to be identical to the number of features we base our prediction on.

However, this is not the case with the first encoding method we explore – the one-hot encoding. The reason for the difference is the enormously increased dimensionality d. As a result of the one-hot encoding, we get $d = 8010$. Such size for input layer will be terribly slow on the cloud configuration used. Even if that were not the case, feature optimization is a practical concern. To slightly offset this imbalance, the network for the one-hot encoder uses 40 neurons in the input layer.

The “curse of dimensionality” [Bellman, R., Adaptive Control Processes: A Guided Tour] is a term used to refer to such a situation as the one observed in the one-hot encoder: the input space is huge. The neural network must be configured in such a way as to guarantee the appropriate mapping between this input space and the output space. In our case the output space is tiny. Additionally, the huge input space presupposes having more data in order to compensate the dimensionality with the number of samples. In such a way, the neural network will be able to more easily determine what is important and what not.

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. ..” [Blum, A., Neural Networks in C++]
- it should never be more than twice as large as the input layer. [Berry, M.J.A.,Data Mining Techniques]

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

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

Table 4. Neuron count per hidden layer

Table 4. displays the number of units used per each setup. The entity embeddings encoding approach requires a lot more units to be effective.

2.2.3. Count of layers in the network

There is hardly any rule or function that we can use to decide how many hidden layers to use in our model. We cannot determine the count of layers without consulting the universal approximation theorem which states that a feed-forward network, with a single hidden layer, containing a finite number of neurons, can approximate continuous functions with mild assumptions on the activation function. [Hornik, K, Multilayer feedforward networks are universal approximators] The problem with this theorem, however, is that it does not specify how hard it will be for a single-layer network to approximate the output. Because the dataset in this case is relatively simple (not related to computer vision for example) 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.

2.2.4. Optimizer

Training the neural network involves a continuous numerical optimization of a nonlinear objective function.

Let:

- C be a matrix of weights or other parameters
- D be a notation for a data set, including both the input and target values:
  - D_T signify the training (learning) set;
  - D_V signify the validation set;
  - D_S signify the test set.
- $TQ(D, C)$ be the total error function

The purpose of the objective function part of any neural network is to minimize $TQ(D, C)$ for the training set. The optimizer must minimize this value. For our setup we used the Adam optimizer. [Diederik P. Kingma, Adam: A Method for Stochastic Optimization] The default parameters follow those provided in the original thesis:

```
keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999,
amsgrad=False)
```

2.2.5. Batch size

The batch size is a hyperparameter of the gradient descent which controls how many training samples to be processed before the model's parameters are updated. In practice, "batch size is typically chosen between 1 and a few hundred […], with values above 10 taking advantage of the speedup of matrix-

matrix products over matrix-vector products.” [Bengio, Y., Practical recommendations for gradient-based training of deep architectures]

What are the implications of using different batch sizes? The batch size b is always $1 < b < n$**.** If b is closer to 1, that means very few test cases are being processed per each iteration of the network. If b is closer to n , then that means we are trying to process the whole dataset at once which can carry a huge memory expense and be very inefficient due to very few weight values updates. Therefore, the batch method according to this definition is called “mini-batching”. It allows for working with different pools of the samples for each network iteration.

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. The number 0.6 which denotes the portion of the data set used for training, is further explained in section 3.1.11.

2.2.6. Epoch count

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

How does the count of epoch affect the model and prediction quality? In a certain way, it is a temporal concept. Given enough time, the network will learn the best approximation. However, given infinite time, i.e. if epochs $\mathbf{e} \rightarrow \infty$**,** 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 behaviour very early in the training process. The second training round is carried out through the course of 100 epochs. This longer training time allows the model to progress and provide a better overview of the learning process.

2.2.7. Loss function

Let

- $M(X, W)$ be the output function computed by the network (the letter M is used to suggest "mean", "median", or "mode")
- $p = M(X, W)$ be an output, or the predicted value for case duration
- y be a target scalar, in this case - case duration
- r be a residual value
- $Q(y, X, W)$ be the case-wise error function written to show the dependence on the weights explicitly
- $L(y, p)$ be the case-wise error function in simpler form where the weights are implicit (the letter L is used to suggest "loss" function)

For any case denoted as ***j*,** the difference between the target value and the output value is $\boldsymbol{r_j = y_j - p_j}$, this is the so-called "residual" or "error"[3].

Error functions are defined so that bigger is worse. Usually, an error function $Q(y, X, W)$ is applied to each case. The function is defined in terms of the target and output values $Q(y, X, W) = L\big(y, M(X, W)\big) = L(y, p)$. Error functions is what we refer to here as the "loss" functions. The error function for the entire data set is usually defined as the sum of the case-wise error functions for all the cases in a data set:

$$TQ(D, C) = \textbf{sum}\left(Q\left(y_j, X_j, W\right)\right)$$

[3] Neural Network FAQ, ftp://ftp.sas.com/pub/neural/FAQ.html

Equation 1. Error function summation

Thus, for squared-error loss, the total error is the sum of squared errors (i.e., residuals), abbreviated SSE. In regressions scenarios it is often more convenient to work with the average (i.e, arithmetic mean) error:

$$AQ(D,W) = TQ(D,W)/D$$

Equation 2. MSE

For squared-error loss, the average error is the mean or average of squared errors abbreviated MSE. Regression models rely heavily on the MSE metric to gauge their performance.

2.2.8. Regularizers

A well-established approach to improving the generalization powers of a mode is regularization, i.e., trying to minimize an objective function that is the sum of the total error function and a regularization function. The regularizes attach a penalty value to the weights of the neural network. This penalty serves as the regularization value that ultimately reduces the chance of overfitting. [Hawkins, D. M., The Problem of Overfitting, Oct 2003]. In turn, we want to avoid the effect of overfitting, which is the case when the neural network parameters have fit too closely to the data.

There exist two types of regularization – L1 and L2. In general, L1 can cut some parameters all the way to 0.0, while L2 only "prunes", i.e. adjusts but never sets weights to 0.0.[4]

2.2.9. The form of the output unit

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.

[4] https://docs.microsoft.com/en-us/archive/msdn-magazine/2015/february/test-run-l1-and-l2-regularization-for-machine-learning

2.2.10. Method-specific hyperparameters

In this section we explore the hyperparameters that some of the encoders require in addition to the baseline hyperparameters.

2.2.10.1. 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. This hyperparameter is further explained in 3.2.3.2.

2.2.10.2. Hashing encoding vector size

The hashing encoding approach involves another extra parameter n referred to as the number of components or vector output size.[5] We perform two runs with $n = 15$ and $n = 50$.

2.2.10.3. Entity embeddings embedding dimension

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.

2.2.11. Training/validation/test split

The training, validation and test split are another set of hyperparameters that have an impact on the model's performance. There are two major points that we must consider when performing the split. On the one hand, the less training

[5] https://contrib.scikit-learn.org/categorical-encoding/hashing.html

data we have, the more variance we introduce in the parameter estimate of the model and ultimately the training error. On the other hand, the less training data we have, the more variance there will be present in the final performance metric, which is the training error.

For our experiment we follow the Pareto distribution [Arnold, B. C., Pareto Distributions, 2nd edition] principle: a ratio of 60/20/20 split among the training, validation and test datasets. In literature, a number of statistical approaches have been introduced for calculating this split such as [Guyon, I., A scaling law for the validation-set training-set size ratio] and [Guyon, I., Makhoul, J., Schwartz, R., Vapnik, V., What Size Test Set Gives Good Error Rate Estimates?] However, we find none of these practically applicable to this regression problem.

Our train and validation data set represent 80% of the whole dataset, while the test dataset is 20% of the whole.

2.3. 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). As we will show, these plots reveal important information about how representative the data set is and how the model can score overall.
    a. **Training dataset loss curve** represents how well the model is learning.
    b. **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. Finally, 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 is expressed in the following way:

$$\Delta G_l = G_{l100} - G_{l10}$$

Equation 3. Proposed measure ΔGl

It indicates if the generalization gap is decreasing or increasing as the training proceeds. Ideally, we are looking for negative or values close to 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.[Jiang, Y., Krishnan, D., Mobahi, H., Bengio, S., Predicting the Generalization Gap in Deep Networks with Margin Distributions] 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 that measures the difference between the validation mean squared valued (MSE) and the training MSE:

$$G_l = MSE_{Validation} - MSE_{Training}$$

Equation 4. Generalization gap equation

- $G_l > 0$ 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. As we will see this scenario does occur sometimes under some encoding methods. This 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, 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.

The MSE, mentioned already in 2.2.7., is a measure according to the formula [James, G., Witten, D., Hastie, T., Tibshirani, R., An Introduction to Statistical Learning 2017]:

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

Equation 5. MSE

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 all train, validation and test datasets.

3. Inputs

3.1. Numerical Input

3.1.1. Standardization

For the natively numerical feature values, as well as the outputs from the hashing encoding, value standardization is applied using the zscore method from scipy stats package. It computes the z score of each value in the sample, relative to the sample mean and standard deviation.

The motivation for standardizing the feature vectors is to ensure that the contribution of each feature value is as equal as possible to all other values. In cases where no standardization is applied, the neural network can become insensitive to features with values that belong to broad ranges.

The target variable “case duration” is also standardized to a mean of 0 and a standard deviation of 1. In this case, this is more convenient because the initial weights of the network will be optimal as the input values and the target will be much more numerically conditioned between each other. Conversely, we would require a greater number of epochs in order to reach a good prediction (optimal weights and low loss).

It is important to note that standardization is applied across the column vectors, not the row vectors. Standardizing over each case (row) will destroy the uniqueness of the values in their context and make this dataset useless for the experiment.

3.2. Categorical Input

3.2.1. Encoding methods

3.2.1.1. One-hot

The one-hot encoding method is probably the most straightforward way to encode categorical features. It is based on a representation of the cardinality of each feature. For each unique value in each feature $\boldsymbol{C}$, we get a new feature $\boldsymbol{C_i}$ that will contain either a 0 or a 1 for the corresponding case. As we will see, this method leads to a drastic increase in the dimensionality of the dataset with features if high cardinality. Even more, the one-hot encoder fails to capture any potential informative relationships between the values – every value is treated independently. [Guo, C., Berkhahny, F., Entity Embeddings of Categorical Variables, April 2016] The following table shows the cardinality of the features subject to encoding:

$\boldsymbol{C}$	**Cardinality**
caller_id	5245
sys_updated_by	845
location	255
category	59
subcategory	255
u_symptom	526
cmdb_ci	51
impact	3
urgency	3
priority	4
assignment_group	79
assigned_to	235

problem_id	253
vendor	5
resolved_by	217
Total:	**8005**

Table 5. Count of unique values per each categorical variable C

The one-hot encoder results in a dimensionality $d = 8010$, comprising the new 8005 features and the 5 natively numerical features. The target variable is excluded from this count. This is a drastic increase of over 400 times from the initial dimensionality $d = 20$. This increase also poses other problems such as the fact that more data samples may be needed for an efficient training process.

3.2.1.1.1. Metrics

The result from the one-hot encoder approach is two very divergent curves for the training loss and validation loss curves. While the model is getting trained well on the training set with a continuously diminishing loss, its performance on the validation set is very poor. This fact suggest that the training set does not provide enough information to learn the problem, relative to the validation dataset used to evaluate it.

This phenomenon typically occurs in training datasets that have too few examples as compared to the validation set. In our case, we can conclude that the drastically increased dimensionality of the model – from 20 to 8010 features, means the 85024 cases we process for training may not be enough to build a good prediction result. This encoding approach could be a better choice for features of low cardinality.

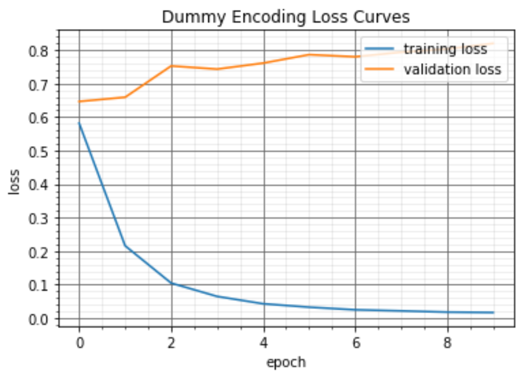


Figure 5. One-hot encoding model metrics, 10 epochs

In fig. 4. we see the initial performance for the first run, which is 10 epochs long. The next plot shows the performance of the model for future epochs, up to the 100th:

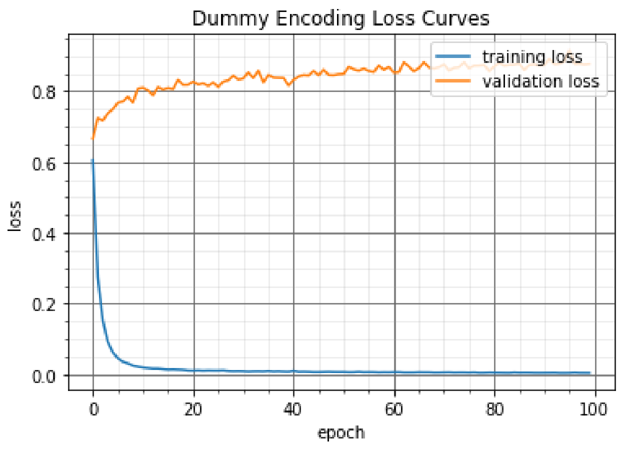


Figure 6. One-hot encoding model metrics, 100 epochs

In future epochs, we observe a flat training loss curve which is a reason for concern. It seems the model is overfitting, i.e. adjusting the parameters too well to the training dataset. Therefore, when presented with the new values – the validation dataset, the model fails to provide a good generalization.

This experiment finishes with the following metrics:

$$G_{l10} = 0.7777$$

$$G_{l100} = 0.8741$$

$$\Delta G_l = 0.8741 - 0.7777 = 0.0964$$

$$L_{train10} = 1.0902;\ L_{train100} = 1.2711$$

Regarding the high value of G_l, the small delta is not indicative of any improvement here. The metrics proove that with time, the predictive capabilities of the model diminish. Even more, the training loss is very high from the start.

3.2.3.1.1. Model Parameters

Layer (type)	**Output Shape**	**Param #**
dense (Dense)	(None, 40)	320440
dense_1 (Dense)	(None, 20)	820
dense_2 (Dense)	(None, 1)	21
Total params:		321,281
Trainable params:		321,281
Non-trainable params:		0

Table 6. One-hot encoding model parameters

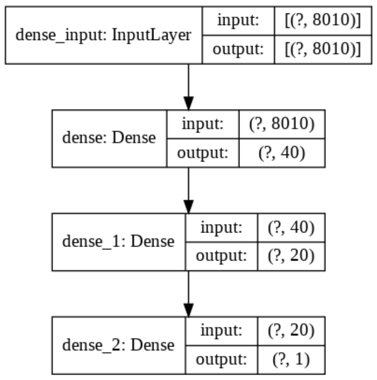


Figure 7. One-hot encoding model visualization

Table 6. shows the total parameter count for the one-hot encoder. The numbers are relatively low. In Figure 7 we can see a visual representation of the sequential model.

3.2.3.2. Target encoding

The idea behind target encoding is to calculate for each $\boldsymbol{c} \in \boldsymbol{C}$ an average of the corresponding values in the target variable y. Then we are going to replace each c with the according mean. [Halford, M., Target Encoding Done The Right Way]. This is done by following these steps[6]:

- The mean for the target value is calculated over the whole dataset
- The count and mean of the target value are calculated for each $c \in C$, i.e. a grouping is performed on each c
- The smoothed mean is calculated according to this formula:

$$\mu = \frac{n \times x + m \times w}{n + m}$$

Where:

- μ is the mean we are computing. It will be the value that replaces the categorical value c
- n is the number of values we have per each group
- x is the estimated mean per group
- m is the "weight" you want to assign to the overall mean
- w is the overall mean.

3.2.3.2.1. Metrics

After applying the target encoding approach to the data, the model produces the following metrics:

[6] Heaton, J. Applications of Deep Neural Networks, Washington University, p. 2.2. Categorical values, link

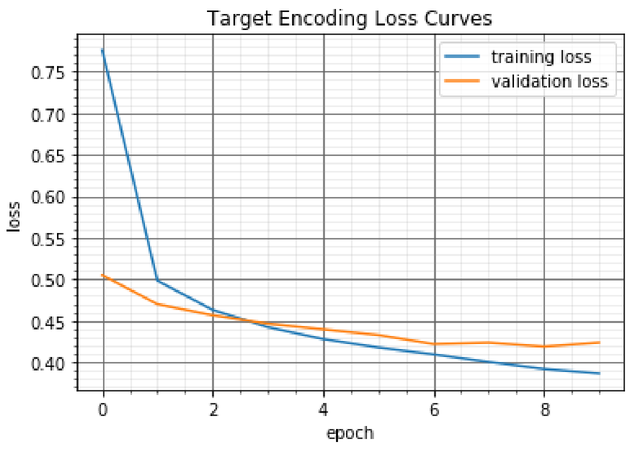


Figure 8. Target encoding model metrics, W=5

In fig. 8. we see the target encoding training and validation loss are pointed in one direction. This fact suggests potential future convergence to one point or a stabilized difference between the two (generalization gap). Both observations suggest that the model can generalize in a good way based on the training data provided.

We exploit the model further by running it for 100 epochs. Then plotting the loss curves again:

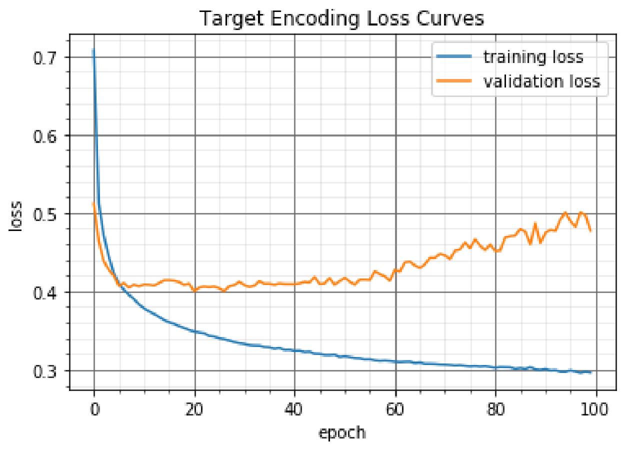


Figure 9. Target encoding model metrics, future epochs, W=5

The plot in fig. 9. confirms our expectation - given enough time, the model can provide a good prediction due to the constant and small generalization gap[7], however only up to epoch 40 to 50. The unaffected dimensionality and good statistical representation of the categorical features are the two factors that contribute to this result. In epochs following epoch 50, the model has indications to perform not as good. This is also a proof of the fact that longer training does not always guarantee better results.

How about if we modify the weight hyperparameter? The plots then appear slightly more promising:

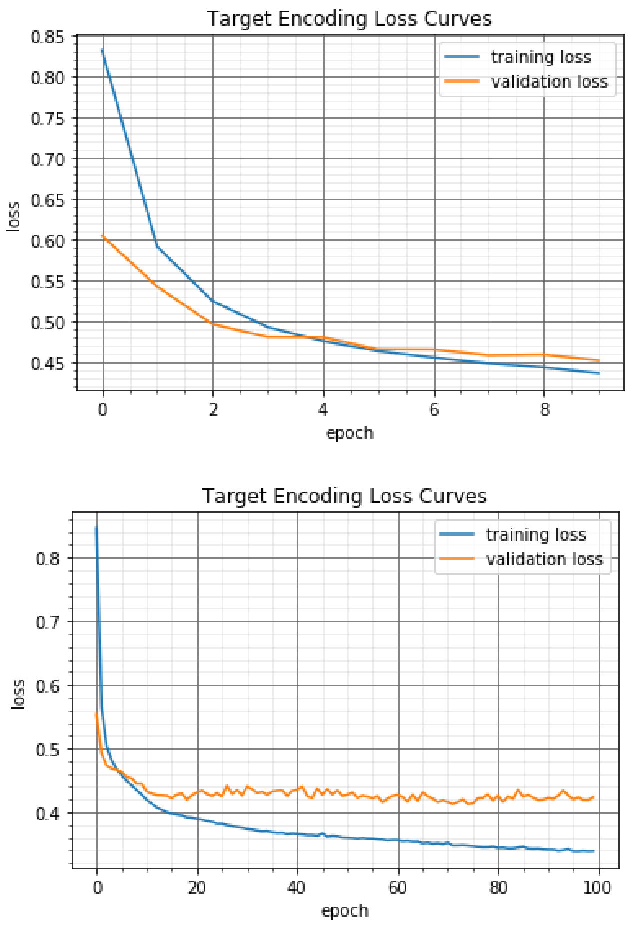


Figure 10. Target encoding model metrics, all epochs, W=100

In fig. 10. we clearly observe that both initially and for longer periods of time, the model performs more efficiently with a weight W = 100. Generally, a

[7] https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

lower weight causes more overfitting. With W = 100, we are declaring that we need at least 100 values for the sample mean to overtake the global mean as defined in 2.2.10.1. This provides further "smoothing" to the encoder and ultimately values for the different features that contribute more efficiently to approximating the output.

The model finishes with the following results:

Epoch	**W**	$\boldsymbol{G_l}$	$\boldsymbol{\Delta G_l}$	$\boldsymbol{L_{train10}}$	$\boldsymbol{L_{train100}}$
10	5	0.0464		0.5366	
100	5	0.1839	0.1375		0.5618
10	100	0.0225		0.6064	
100	100	0.0874	0.0649		0.6708

Table 7. Model final metrics

The model produces acceptable results prior to the 100th epoch if the target weight is 100. What further positive changes we could produce with a different weight value is a subject of more experiments. It is also heavily dependent on the source data.

3.2.3.2.2. Model Parameters

Layer (type)	**Output Shape**	**Param #**
dense (Dense)	(None, 20)	420
dense_1 (Dense)	(None, 10)	210
dense_2 (Dense)	(None, 1)	11
Total params:		641
Trainable params:		641
Non-trainable params:		0

Table 8. Target encoding model parameters

Table 8. demonstrates that the model has a very low number of parameters as compared to the one-hot encoding. The dimensionality has been preserved to a low value of 20.

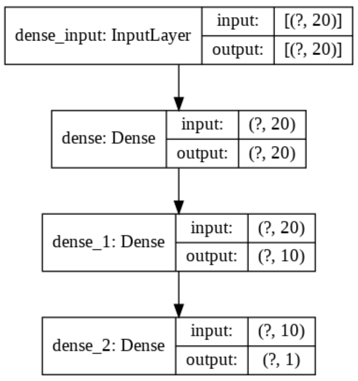


Figure 11. Target encoding model visualization

The model is identical to the one previously seen in the one-hot encoding.

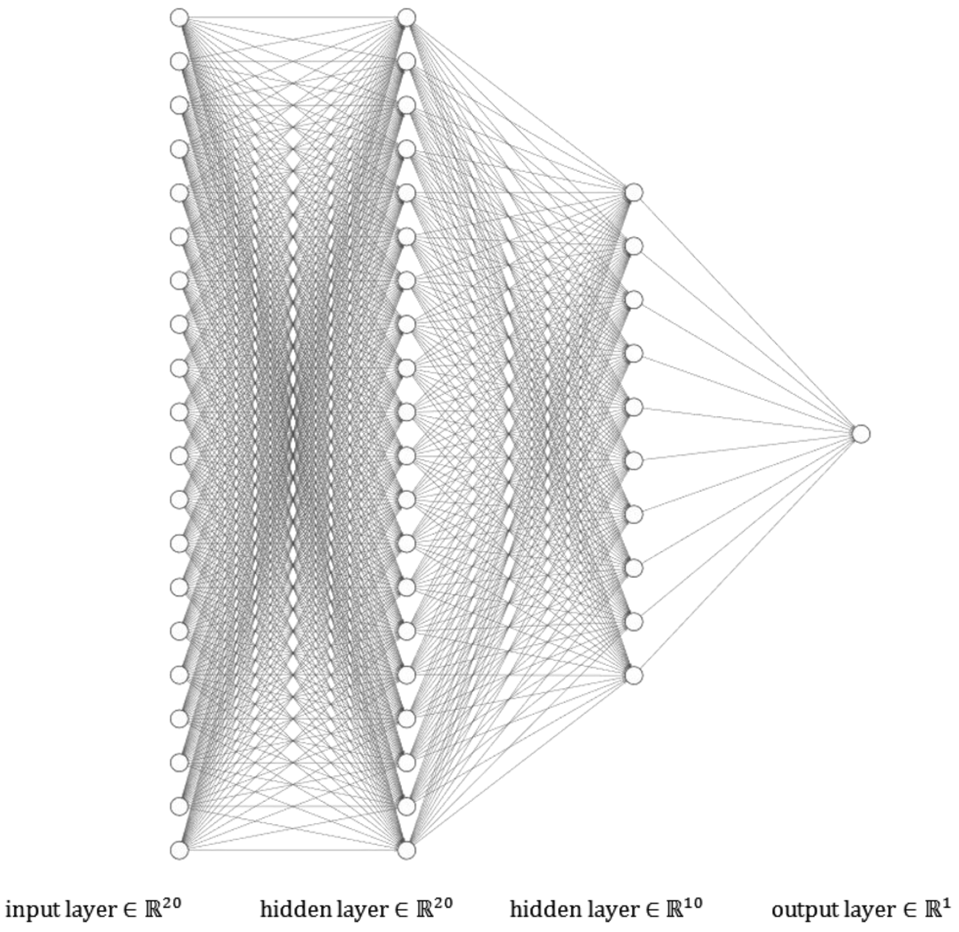


Figure 12. Target encoding full architecture schematics

In Fig. 12. we see a “birds-eye” view of the model. For practical reasons, the schematics of the 8010 (one-hot) and 121 (binary) encoding models will not be shown in subsequent sections.

3.2.3.3. Binary

The binary encoding method is a variation of the one-hot encoding. The aim here is to convert each categorical feature vector ***C*** into its binary representation after substituting each value with its ordinal representation. This process includes the following steps:

1. Assign ordinal numbers to the values in the vector
2. Convert those numbers to their binary representations
3. Extract the new feature vectors

For example, let us take the ***category*** feature from the existing dataset. It is one of the categorical features that we want to encode. The values from the first 3 rows of the data set look like this:

category
category 1
category 2
category 3

First, these values will be converted to ordinal numbers: [Potdar, K., Pardawala, T., Pai, C. A Comparative Study of Categorical Variable Encoding Techniques for Neural Network Classifiers]

category
1
2
3

Then the numbers will be converted to their binary representations:

category
01
10
11

The results look like this:

category_0	category_1
0	1
1	0
1	1

Overall, the category input vector has a cardinality of 59: $|C_{category}| = 59$. The following formulas for the resulting feature count F_C can be deduced for the binary encoder as compared to the one-hot encoder[8]:

- Binary:
  - $|C_{category}|$, in this case: $59 \approx 6$
- One-hot:
  - $F_C = |C_{category}|$

So high cardinality has a limited impact on the binary encoding. It is only a logarithmic function of the input feature cardinality.

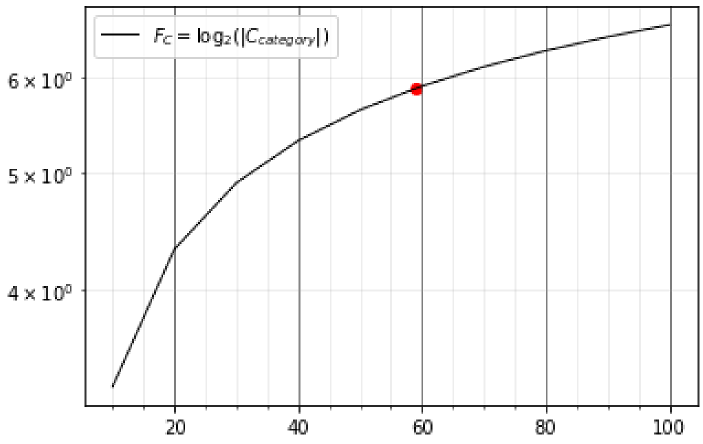


Figure 12. Binary encoder feature count: the x axis is the cardinality; the y axis is the base-2 logarithm of the cardinality

$x = 59, y \approx 5.88$

[8] https://stats.stackexchange.com/questions/325263/binary-encoding-vs-one-hot-encoding

Fig. 12. demonstrates the plot of a logarithm with base 2 taken from the feature cardinality.

3.2.3.3.1. Metrics

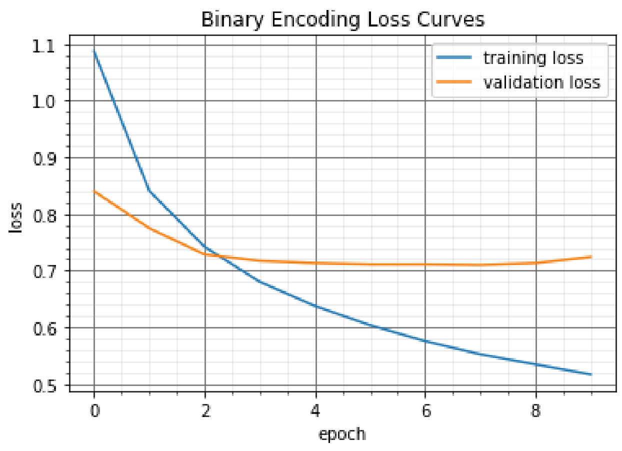


Figure 13. Binary encoding model metrics, 10 epochs

On the 10-epoch graph in fig.13. we can examine the steadily decreasing training loss. On the other hand, the validation loss reaches a low point at about 0.72 MSE and remains close to a flatline from epoch 4 onward. In such a case, it is also worth extending the running time of the model to 100 epochs, in order to get a better overview of the curves in the future. At epoch 20, on fig. 14., one could think the generalization gap would remain steady or even possibly converge. However, this is not the case. From epoch 30 on, the validation loss curve begins growing. This is further made clear in future epochs – the 60th, 80th and all the way to the 100th. The training episode terminates with diverging training and validation loss curves suggesting that the increased dimensionality has again impacted the learning ability of the network.

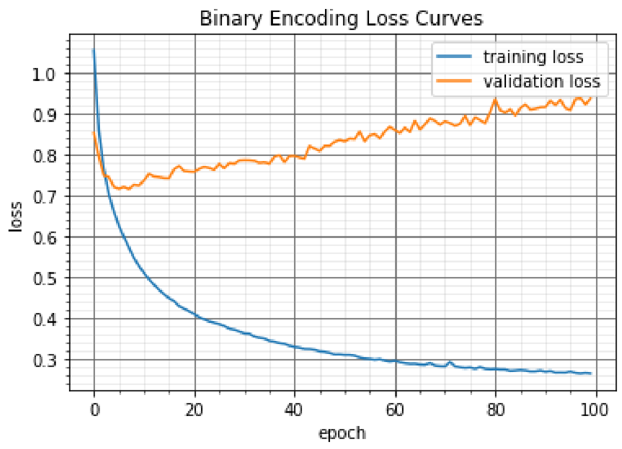


Figure 14. Binary encoding model metrics, future epochs

The resulting metrics are:

Epoch	G_l	ΔG_l	$L_{train10}$	$L_{train100}$
10	0.2096		0.7429	
100	0.6739	0.4643		1.6971

Table 9. Model metrics

3.2.3.3.2. Model Parameters

Table 10. shows the parameter counts that have been established for this model:

Layer (type)	**Output Shape**	**Param #**
dense (Dense)	(None, 20)	2440
dense_1 (Dense)	(None, 10)	210
dense_2 (Dense)	(None, 1)	11
Total params:		2661
Trainable params:		2661
Non-trainable params:		0

Table 10. Binary encoding model parameters.

The network topology as shown in in fig. 14., has stayed the same:

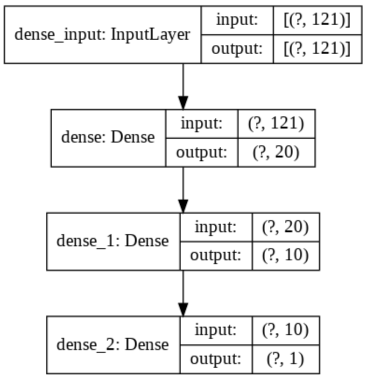


Figure 15. Binary encoding model visualization

3.2.3.4. Hashing

The hashing encoding approach for handling categorical data involves using a hash function to map feature values to indices in the resulting feature vector. The result is that after hashing, we just get the location in the vector we have defined by the vector size hyperparameter function. This approach is very suitable for feature vectors of high cardinality such as the ***caller_id*** and ***sys_updated_by*** features.

One of the major disadvantages of the method is that reverse mapping from the index to the real value is not possible. In our scenario this option is not required. A more concerning subject, however, could be the problem of potential hashing collisions. Different values from the input feature vector could be mapped to one and same value in the resulting feature vector. To avoid such collisions, we use the SHA256 hashing function. Its probability of running into a collision for the ***caller_id*** feature (which has the highest cardinality of 8145 in the dataset) can be calculated by using the following formula:

$$P_{collision} = \frac{p^2}{2^{n+1}}$$

Where $p = 8145$ and $n = 255$, hence:

$$P_{collision} = \frac{8145^2}{2^{255+1}} \cong \frac{6.6 \times 10^7}{10^{77}} \cong 6.6 \times 10^{-70},$$

which is a value of extremely low significance.

Considering the facts so far, the overall flow of the hashing encoder looks like this:

1. Take input feature value, e.g. "Category 55"
2. Hash value
   Sha256("Category 55") =
   bebba9921075d9a9893029024fb663506ec7ad9a743659ac6fd3fde8e58c6fd7
3. Convert from the hexadecimal to the integer representation[9]
   int(hasher.hexdigest(), 16)
4. Take the modulo of the result and the vector size hyperparameter
   int(hasher.hexdigest(), 16) % 15

For this encoder, the biggest challenge is to decide on the value of the output vector. There is no set formula for choosing this value, although some authors suggest establishing a dependency on the cartesian product of the features being encoded[10]. This approach appears to be practical for features with low cardinality.

For our purposes, we conduct two runs of the model with the hashing encoder. The first one with encoded vector of size 15, i.e. equal to the starting dimensionality of the categorical features. The second one is with a vector of

[9] https://contrib.scikit-learn.org/categorical-encoding/_modules/category_encoders/hashing.html#HashingEncoder.hashing_trick

[10] https://pkghosh.wordpress.com/2019/08/07/encoding-high-cardinality-categorical-variables-with-feature-hashing-on-spark/

size 50. The following table provides an overview of the actual difference in the resulting encoding:

string	Hashed result	Int representation	Resulting index (%15)	Resulting index (%50)
Category 9	8fb39ab2e53213eeb63d7c75b33cfe7c030d70a794e86f214ccc0ce4fbd75724	649980706639474524613255945553345982558186719553941199688270740888721077921 64	4	14
Category 20	9e0d81fcaba150a5e640ced7f14155d2ba1f281c0d5f7596c268638af6aad86b	714892962252582674941261830056188080307943628098692663056540102542087596831 79	14	29
Category 40	f324c56964d5ac196dd7fc48a308d79bf1c829ee95bde4c5db3ac460f02d3bea	109976991185509600839963661068367643614311817999712487004505233179993518717930	5	30
Category 54	dbf82be3cab784c11ace08ca25a287f3f80157542562656bed128fc0e1900171	dbf82be3cab784c11ace08ca25a287f3f80157542562656bed128fc0e1900171	12	27
Category 60	dbf82be3cab784c11ace08ca25a287f3f80157542562656bed128fc0e1900171	102886594802262606587397109971840769413423046789579414069500732125200050946170	0	20

Table 11. Hashed and encoded representations

The table illustrates the resulting index value depending on the output vector size – 15 or 50. Finally, the array element at the corresponding index is increased by 1 for each encounter of the value for thus providing the final integer representation of the categorical value.

3.2.3.4.1. Metrics

Here we present two sets of metrics due to the rather significant impact the size of the resulting vector has on the model performance. In table 12, the result from two runs with 10 and 100 epochs are presented with vector size 15.

In the subsequent table 13, the results from another two runs are given this time with a vector size of 50. Furthermore, since the hashing encoder produces a single integer representation for the corresponding string value, we have considered applying the zscore standardization for the values. In this way, we can compare the performance between standardized and default encoded values.

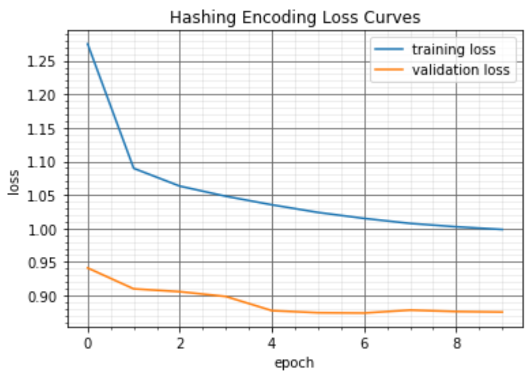
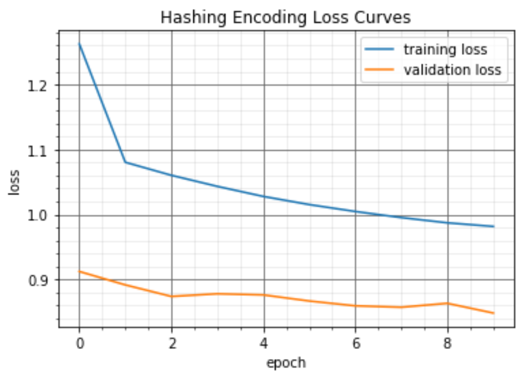
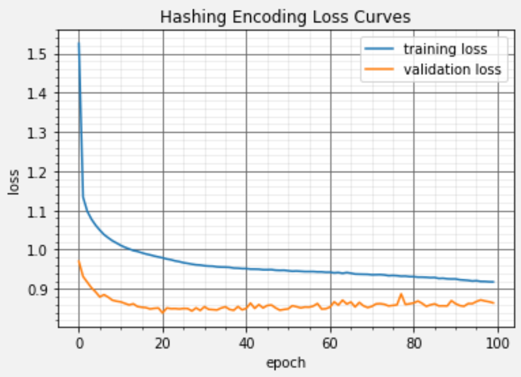
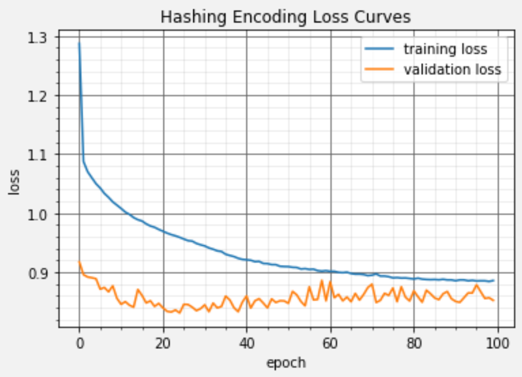
Epochs	**Standardized, v=15**	**Default, v=15**
10		
100		

Table 12. N components = 15, all epochs with standardized and default encoded values

In Table 12. we unambiguously observe that the training loss is always higher for all model variations, regardless of vector size or value standardization. This is a special scenario in deep learning, indicating a validation data set that is "easier" for the model than the training dataset. This fact is surprising because the validation dataset is unknown to the model. Therefore, how is it possible to get better prediction on it, as compared to the training dataset? In this case we see that the model interprets the data differently. This could mean adjusting the data split. The performance is not influenced much by standardized and default versions of the hashed data representation as well as by the vector size.

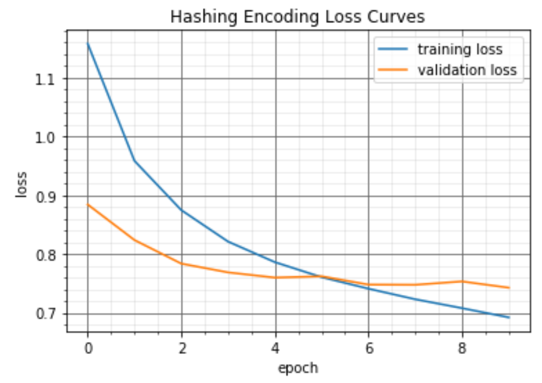
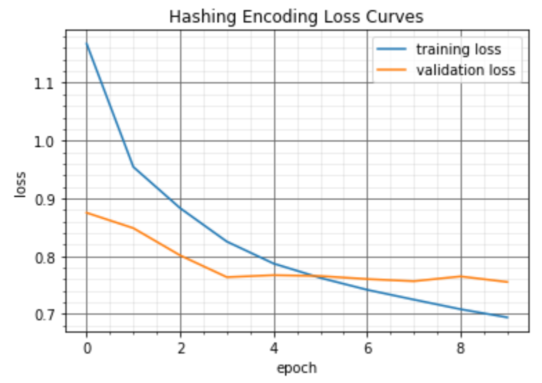
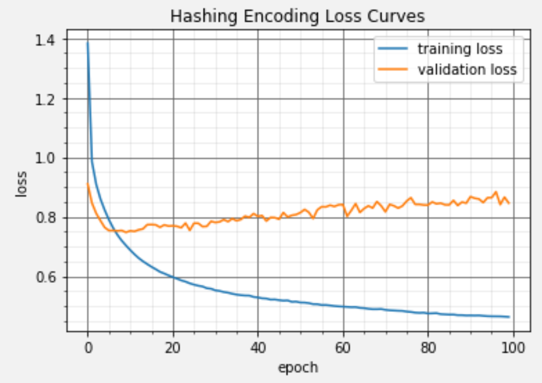
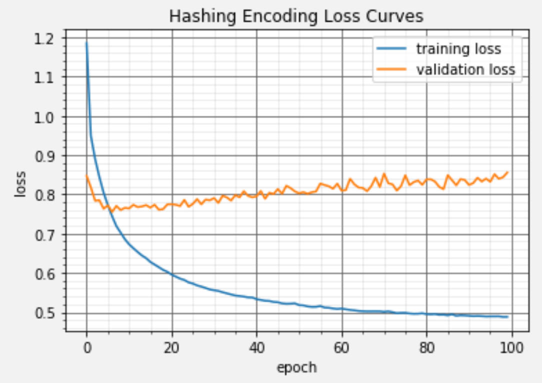
Epochs	Standardized, v=50	Default, v=50
10		
100		

Table 13. N components = 50, all epochs with normalized and default encoded values

In table 8, the increased dimensionality to 50 clearly provides different and better results up to a certain extent. On the one hand, we have the normalized data representation. There is a clear stabilization of the training and validation curves to a point that makes them parallel to each other. The generalization gap indeed has remained higher at about 0.2 MSE suggesting a possible overfit.

On the other hand, the default data exhibits an even larger generalization gap. Even more, the validation loss curve is growing consistently along with the model experience, which is a proof of an overfitting model.

In summary, we can state that the hashing encoder performs well for these data only if:

- Output vector dimensionality is slightly increased to a value between d and 2 x d;
- Resulting hashed index value representations are not standardized.

The final metrics are:

Epoch	**N**	**Standardized?**	G_l	ΔG_l	$L_{train10}$	$L_{train100}$
10	15	no	-0.1268		0.7306	
100	15	no	-0.0315	0.0953		0.7331
10	50	no	0.0663		0.7360	
100	50	no	0.3685	0.3022		0.8511
10	15	yes	-0.1165		0.7538	
100	15	yes	-0.0516	0.0649		0.7419
10	50	yes	0.0545		0.7642	
100	50	yes	0.3834	0.3289		0.8517

Table 14. Final metrics

Overall, the model with hashing vector size of 50 performs best with default non-standardize values. Standardization seems to affect negatively the predicative powers of the model, regardless of the vector size suggesting that the hashing approach is a good way to standardize the data, without extra methods applied.

3.2.3.4.2. Model Parameters

We also get two sets of parameters, one for each model.

	v = 15		**v = 50**	
Layer (type)	Output Shape	Param #	Output Shape	Param #
dense (Dense)	(None, 20)	420	(None, 20)	1120
dense_1 (Dense)	(None, 10)	210	(None, 10)	210
dense_2 (Dense)	(None, 1)	11	(None, 1)	11
Total params:		641	**Total params:**	1 341
Trainable params:		641	**Trainable params:**	1 341
Non-trainable params:		0	**Non-trainable params:**	0

Table 15. Hashing encoding model parameters.

Table 15. show the natural difference in parameter count depending on the output vector size.

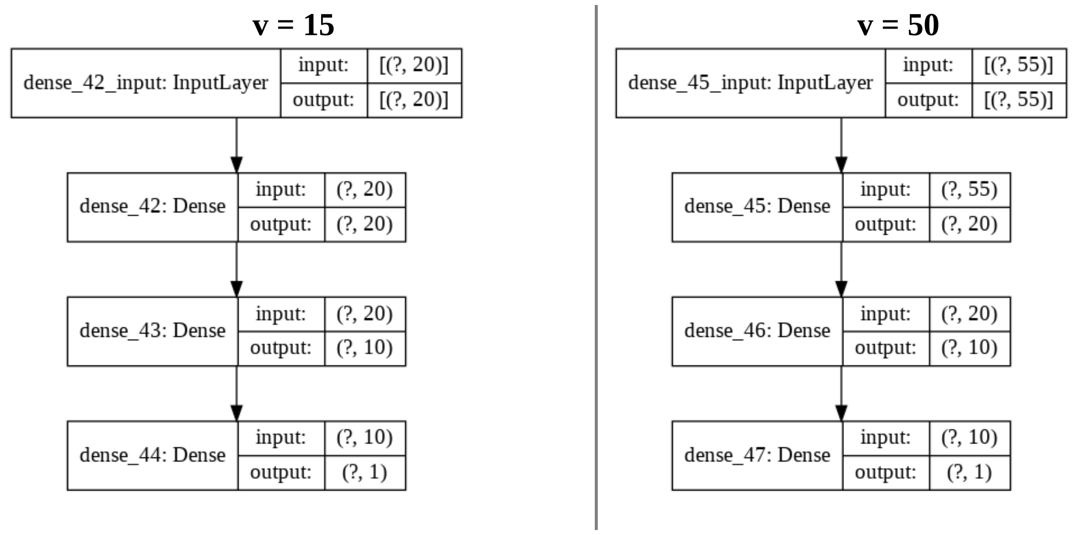


Table 16. Hashing encoding model visualization.

In table 16. we see head-to-head how similar the models are. The only difference is the corresponding input vector size.

3.2.3.5. Entity Embeddings

The entity embeddings approach for encoding categorical variables is in a category of its own, as compared to the rest of the methods described so far. The reason is that it requires a different network topology based on the feature cardinality. Importantly, one input and one embedding layer are needed for each encodable value. Therefore, the required count of those layers depends on the features we must encode, in this case 15 categorical and one embedding layer for all numerical features.

The embedding is a much more efficient approach than the one-hot encoding, for example. Even more, it allows for properly capturing the relationships between the different variables [Guo, C., Berkhahny, F., Entity Embeddings of Categorical Variables]. The overall process for making an embedding layer for each feature looks like this:

1. We establish and embedding size for each categorical feature with the following formula:

   int(min(np.ceil((no_of_unique_values_in_feature)/2), 50))

   This value is a unique hyperparameter to this setup and is of maximum size 50 or the count of unique features values divided by 2. We will also make another instance of the model with an embedding size of 100.

   According to the embedding dimension we get the following values:

C	**Cardinality**	**EE Dimension**
caller_id	4106	50
sys_updated_by	601	50
location	190	50
category	44	22
subcategory	211	50
u_symptom	471	50
cmdb_ci	45	23
impact	3	2
urgency	3	2
priority	4	2
assignment_group	72	36
assigned_to	196	50
problem_id	131	50
vendor	4	2
resolved_by	176	50

Table 17. Feature cardinality and resulting embedding size.

2. The cardinality of each variable becomes its input dimension, while the embedding size is the output dimension. So here we are trying to mimic a

dimensionality reduction technique in a way – we want to map the larger input space to the more compact output space.

3. Each feature is converted from its default dimensionality of 85024 x 1 into a list of 1 x 85024 values. Each value is the index of the unique values.
4. A multidimensional array is instantiated. Its elements are lists containing the encoded categorical values.
5. This multidimensional array is passed to model. From this input, the input layers are extracted and after them the embedding layers. Ultimately, the matrices of the embedding layers are concatenated. The result is a dense layer with an input of size the sum of all embedding sizes, which is equal to 617.

We get a representation for each categorical feature value that is in the shape of 1 x 50 (the maximum embedding size or the number of unique values divided by 2). We will further explore the model with an embedding size of 100. Let us take as an example the impact feature:

Raw string value	**Embedding vector v**
1-High	$[v_1, v_2]$
2-Medium	$[v_1, v_2]$
3-Low	$[v_1, v_2]$

Table 18. Embedding representation of the "impact" categorical feature

But what are the actual values of v_1 *and* v_2? These values are instantiated randomly at first. As the learning begins and progresses, the network adjusts the values in the matrix based on the gradient descent.[11] The result is a matrix M of size $m \times n$ where $m = 3$ (in the case of the “impact” feature) and $n = 2$, which in this case is the resulting embedding size.

[11] Embedding matrix explained by Andrtwe Ng, link

According to the formula used, the embedding vector is a hyperparameter. The embedding layer is instantiated by using the Keras functional API.[12]

Naturally these 15 matrices from the categorical variables and one for the 5 native numerical features demand a more robust network setup. We use two dense layers, the first with 1000, while the second with 512 units as also mentioned here [Mikolov, T. et al., Efficient Estimation of Word Representations in Vector Space]

3.2.3.5.1. Metrics

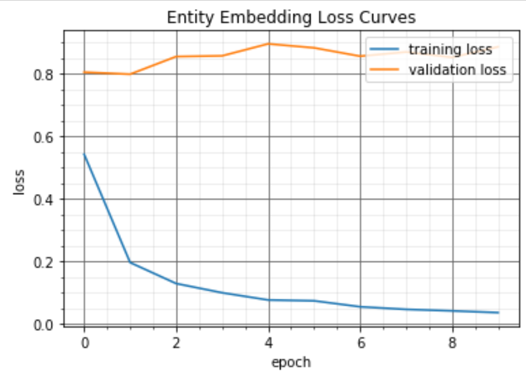
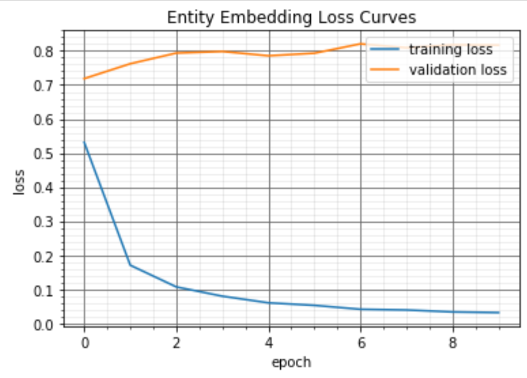
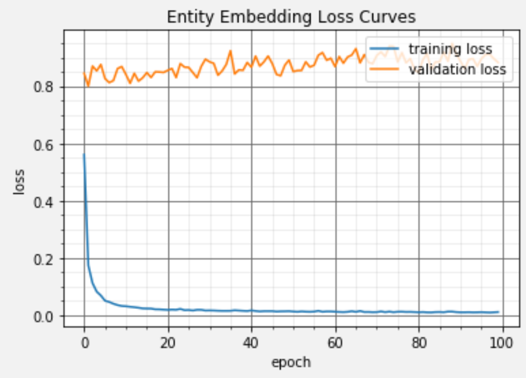
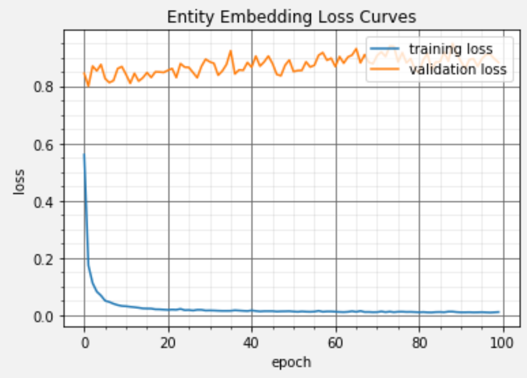
Epochs	Embedding size 50	Embedding size 100
10		
100		

Table 19. Entity embedding metrics

In Table 18. we see observe the two sets of metrics, for the two separate embedding sizes – 50 and 100. In both cases the model overfits, producing a flatline train error curve and very high and consistently divergent validation loss

[12] Embedding layer source code

curve. The results are not optimistic. Choosing the right encoding approach could be a heuristic endeavour.

This encoder finishes with the following metrics:

Epoch	N	G_l	ΔG_l	$L_{train10}$	$L_{train100}$
10	50	0.8500		0.0356	
100	50	0.9702	0.1202		0.0103
10	100	0.7837		0.0334	
100	100	0.8720	0.0883		0.0108

Table 20. Model metrics

3.2.3.5.2. Model Parameters

Layer (type)	Output Shape	Param #	Connected to
input_337 (InputLayer)	(None, 1)	0	
input_338 (InputLayer)	(None, 1)	0	
input_339 (InputLayer)	(None, 1)	0	
input_340 (InputLayer)	(None, 1)	0	
input_341 (InputLayer)	(None, 1)	0	
input_342 (InputLayer)	(None, 1)	0	
input_343 (InputLayer)	(None, 1)	0	
input_344 (InputLayer)	(None, 1)	0	
input_345 (InputLayer)	(None, 1)	0	
input_346 (InputLayer)	(None, 1)	0	
input_347 (InputLayer)	(None, 1)	0	
input_348 (InputLayer)	(None, 1)	0	
input_349 (InputLayer)	(None, 1)	0	
input_350 (InputLayer)	(None, 1)	0	
input_351 (InputLayer)	(None, 1)	0	
assigned_to_Embedding (Embedding)	(None, 1, 50)	9800	input_337[0][0]
assignment_group_Embedding (Embedding)	(None, 1, 36)	2592	input_338[0][0]
caller_id_Embedding (Embedding)	(None, 1, 50)	205300	input_339[0][0]
category_Embedding (Embedding) 968	(None, 1, 22)	968	input_340[0][0]
cmdb_ci_Embedding (Embedding)	(None, 1, 23)	1035	input_341[0][0]
impact_Embedding (Embedding) 6	(None, 1, 2)	6	input_342[0][0]
location_Embedding (Embedding)	(None, 1, 50)	9500	input_343[0][0]

priority_Embedding (Embedding)	(None, 1, 2)	8	input_344[0][0]
problem_id_Embedding (Embedding	(None, 1, 50)	6550	input_345[0][0]
resolved_by_Embedding (Embedding)	(None, 1, 50)	8800	
subcategory_Embedding (Embedding)	(None, 1, 50)	10550	input_347[0][0]
sys_updated_by_Embedding (Embedding)	(None, 1, 50)	30050	input_348[0][0]
u_symptom_Embedding (Embedding)	(None, 1, 50)	23550	input_349[0][0]
urgency_Embedding (Embedding)	(None, 1, 2)	6	input_350[0][0]
vendor_Embedding (Embedding)	(None, 1, 2)	8	input_351[0][0]
input_352 (InputLayer)	(None, 5)	0	
reshape_316 (Reshape)	(None, 50)	0	assigned_to_Embedding[0][0]
reshape_317 (Reshape)	(None, 36)	0	assignment_group_Embedding[0][0]
reshape_318 (Reshape)	(None, 50)	0	caller_id_Embedding[0][0]
reshape_319 (Reshape)	(None, 22)	0	category_Embedding[0][0]
reshape_320 (Reshape)	(None, 23)	0	cmdb_ci_Embedding[0][0]
reshape_321 (Reshape)	(None, 2)	0	impact_Embedding[0][0]
reshape_322 (Reshape)	(None, 50)	0	location_Embedding[0][0]
reshape_323 (Reshape)	(None, 2)	0	priority_Embedding[0][0]
reshape_324 (Reshape)	(None, 50)	0	problem_id_Embedding[0][0]
reshape_325 (Reshape)	(None, 50)	0	resolved_by_Embedding[0][0]
reshape_326 (Reshape)	(None, 50)	0	subcategory_Embedding[0][0]
reshape_327 (Reshape)	(None, 50)	0	sys_updated_by_Embedding[0][0]
reshape_328 (Reshape)	(None, 50)	0	u_symptom_Embedding[0][0]
reshape_329 (Reshape)	(None, 2)	0	urgency_Embedding[0][0]
reshape_330 (Reshape)	(None, 2)	0	vendor_Embedding[0][0]
dense_78 (Dense)	(None, 128)	768	input_352[0][0]
concatenate_22 (Concatenate)	(None, 617)	0	reshape_331[0][0] … reshape_345[0][0] dense_82[0][0]
dense_83 (Dense)	(None, 1000)	618000	concatenate_22[0][0]
activation_38 (Activation)	(None, 1000)	0	dense_83[0][0]
dropout_31 (Dropout)	(None, 1000)	0	activation_38[0][0]
dense_84 (Dense)	(None, 512)	512512	dropout_31[0][0]
activation_39 (Activation)	(None, 512)	0	dense_84[0][0]
dropout_32 (Dropout)	(None, 512)	0	activation_39[0][0]
dense_85 (Dense)	(None, 1)	513	dropout_32[0][0]

Total params:	1,440,516	
Trainable params:	1,440,516	
Non-trainable params:	0	

Table 21. Entity embeddings model parameters.

Table 21. shows the initial network run with an embedding size of 50. The embedding of size 100 results in total trainable params 2,083,708.

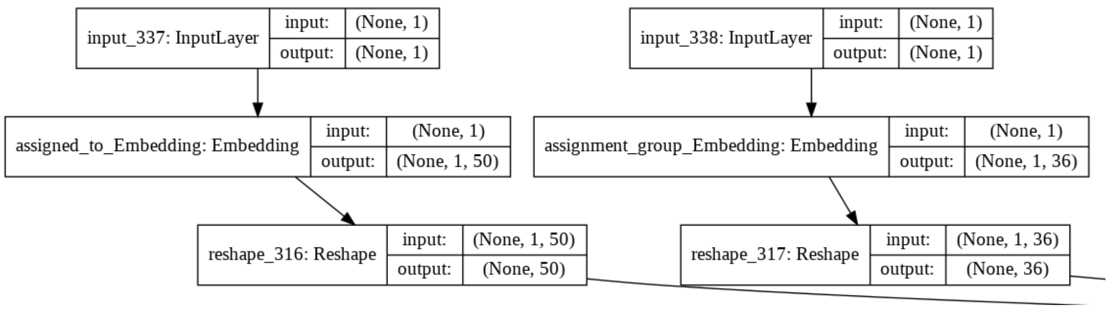


Figure 16. The Input, embedding and reshape layers for the first two categorical variable. The rest are similar.

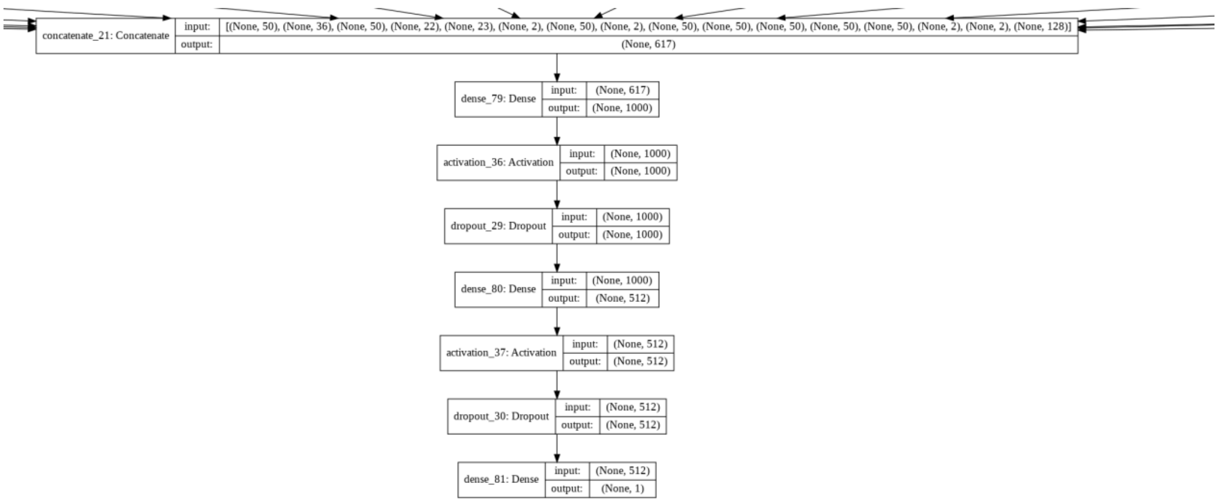


Figure 17. Entity embeddings model visualization. For practical purposes the input and embedding layers are not shown and only the bottom part of the model is depicted where all layers converge.

3.2. Comparative analysis of all methods

3.2.3. Comparative table and summary

The following table 21. displays all metrics for the encoding approaches discussed in this thesis:

Encoding method	epochs	d	$Loss_{train}$	$Loss_{val}$	$Loss_{test}$	Gl	ΔGl	t (h:m:s)
one-hot I	10	8010	0.0177	0.7949	1.0902	0.7777		00:03:16
one-hot II	100	8010	0.0042	0.8776	1.2711	0.8741	0.0964	00:32:31
target I w5	10	20	0.3865	0.4237	0.5366	0.0464		00:00:06
target II w5	100	20	0.2962	0.4772	0.5618	0.1839	0.1375	00:00:53
target I w100	10	20	0.4363	0.4520	0.6064	0.0225		00:00:08
target II w100	100	20	0.3390	0.4236	0.6708	0.0874	0.0649	00:01:03
binary I	10	121	0.5168	0.7240	0.7429	0.2096		00:00:08
binary II	100	121	0.2635	0.9362	1.6971	0.6739	0.4643	00:01:12
hashing I - default	10	20	0.9817	0.8485	0.7306	-0.1268		00:00:07
hashing II - default	100	20	0.8860	0.8525	0.7331	-0.0315	0.0953	00:00:57
hashing III - default	10	55	0.6940	0.7553	0.7360	0.0663		00:00:08
hashing IV - default	100	55	0.4882	0.8556	0.8511	0.3685	0.3022	00:01:08
hashing I - normalized	10	20	0.9985	0.8753	0.7538	-0.1165		00:00:07
hashing II - normalized	100	20	0.9178	0.8643	0.7419	-0.0516	0.0649	00:01:00
hashing III - normalized	10	55	0.6920	0.7426	0.7642	0.0545		00:00:08
hashing IV - normalized	100	55	0.4633	0.8459	0.8517	0.3834	0.3289	00:01:05
embeddings - v50 I	10	20	0.0356	0.8856	1.1423	0.8500		00:00:25
embeddings - v50 II	100	20	0.0103	0.9805	1.2280	0.9702	0.1202	00:03:31
embeddings – v100 I	10	20	0.0334	0.8171	1.070	0.7837		00:23:02
embeddings – v100 II	100	20	0.0108	0.8828	1.1383	0.8720	0.0883	00:03:33

Table 22. All encoding method metrics

Let us summarize each set of network runs for each encoding method.

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 **ΔGl** = 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 fig.18. 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.

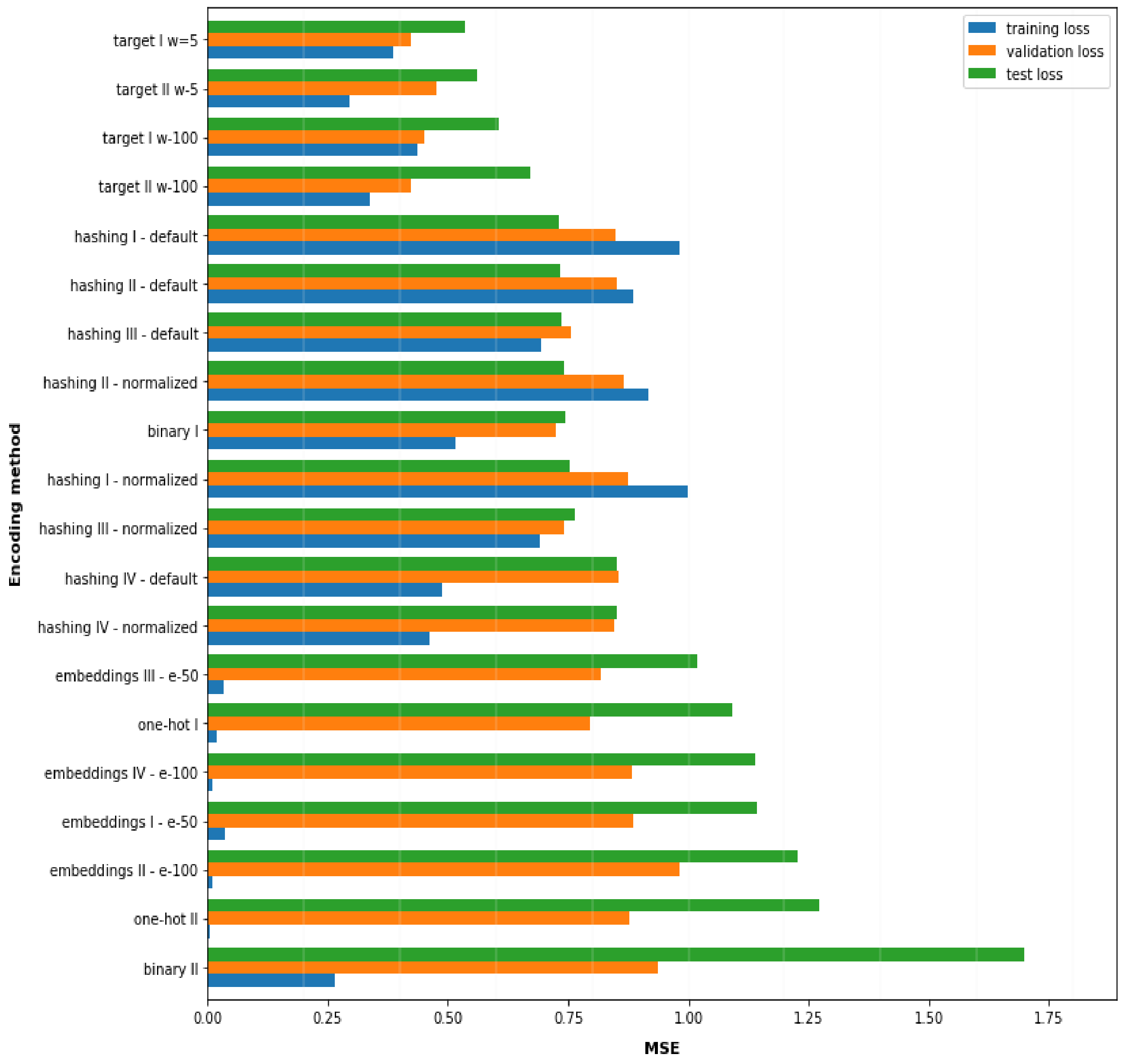


Figure 18. Performance visualization for all models.

In the next plot, we examine the proposed in eq. 3. measure **ΔGI**. All encoding methods are sorted in ascending order by their **ΔGI** values. With this visualization we would like to establish a strong correlation between the best performing method regarding the test loss and the best performing method regarding **ΔGI**. As we have seen in fig. 18., the target encoder exhibits the best performance.

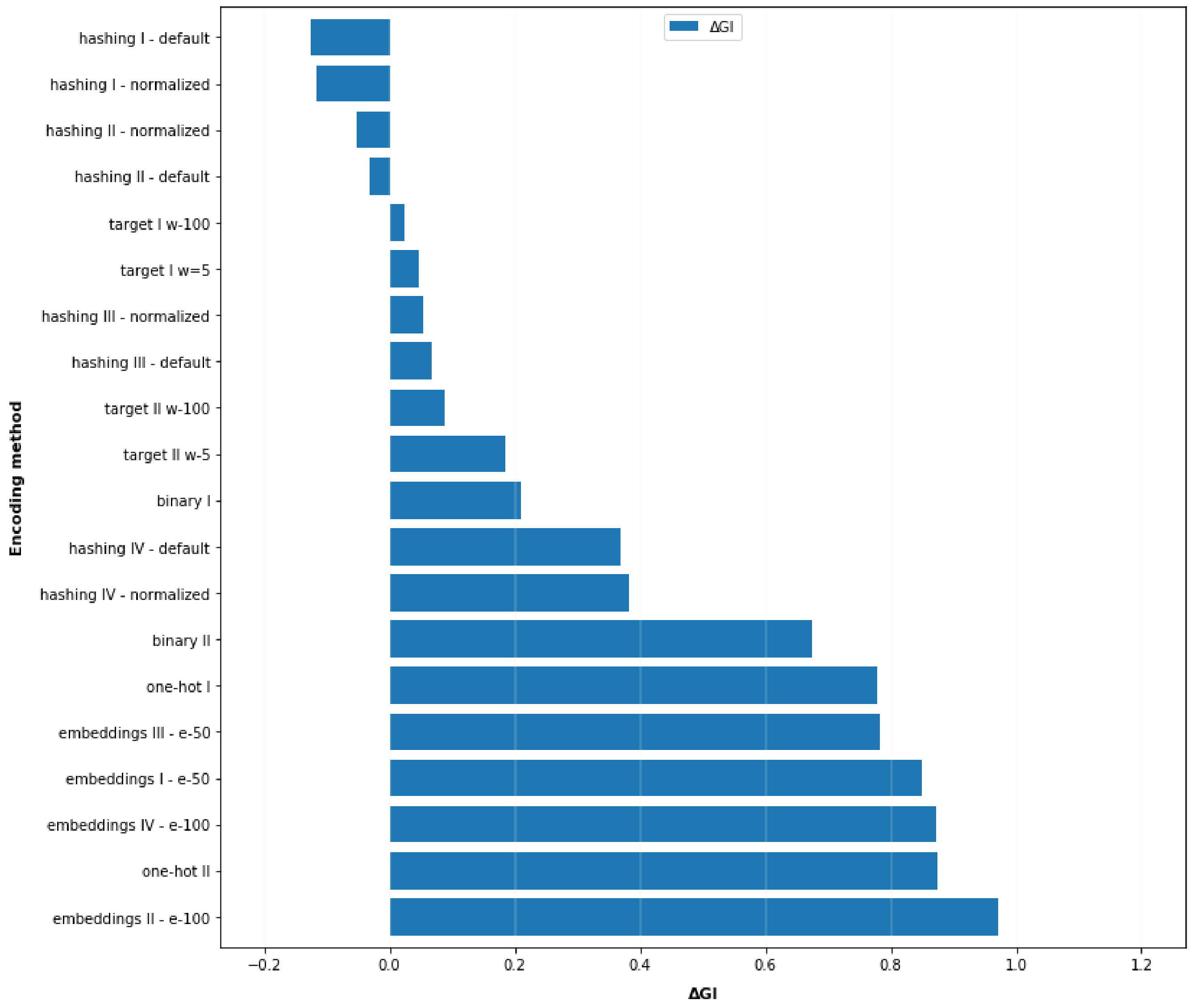


*Figure 19. Plot of **ΔGI** for each encoder*

In fig. 19. we see that the target encoder has one of the lowest **ΔGI** values of all:

- 0.1375 for the set of runs with w=5
- 0.0649 for the set of runs with w=100.

These results are comparatively good based on the other methods’ results as it indicates a convergent generalization gap with time and tuning of the hyperparameter **w**. In short, the result confirms the fitness of the target encoder for the presented dataset and prediction task.

Interestingly, we observe that the hashing encoder with dimensionality d = 20 with both default and standardized values is a leader in terms of **ΔGl.** This finding is also very much in line with the relatively good performance of the hashing encoder as show in fig. 18.

We can conclude that **ΔGl** is a very strong auxiliary metric for determining model performance. Our two best performing encoding approaches in terms of their test loss– the target and the hashing encoding exhibit fittingly good results in terms of their **ΔGl** values.

4. Conclusion

With this work we hope to have established an initial overview of a selected few categorical encoding methods. This work is by no means exhaustive of all the available methods. Many other methods exist that have not been examined here, such as: simple coding, deviation coding, difference coding, Helmert coding, orthogonal polynomial coding, repeated coding[13], 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

[13] https://stats.idre.ucla.edu/spss/faq/coding-systems-for-categorical-variables-in-regression-analysis-2/

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 [Erich Schikuta, E., Neural Networks and Database Systems, University of Vienna]. Even more, neural network modelling and advanced analytics should be considered at the start of any systems design and architecture.

With this work we hope to have contributed to future efforts of making neural network an inherent part of databases and business systems.

Appendix 1: Sources

1. Zurada, J.M. (1992), Introduction To Artificial Neural Systems, Boston: PWS Publishing Company., p. XV
2. Bellman, R. (1961), Adaptive Control Processes: A Guided Tour, Princeton University Press.
3. Blum, A. (1992), Neural Networks in C++, NY: Wiley
4. Berry, M.J.A., and Linoff, G. (1997), Data Mining Techniques, NY: John Wiley & Sons.
5. Hornik, K., Stinchcombe, M. and White, H. (1989), Multilayer feedforward networks are universal approximators, Neural Networks, 2, 359-366.
6. Diederik P. Kingma, Jimmy Ba, Adam: A Method for Stochastic Optimization, 22 Dec 2014
7. Diza, G., Fokoue, A., Nannicini G., Samulowitz, H., An effective algorithm for hyperparameter optimization of neural networks, 23 May 2017
8. Goodfellow , I., Bengio, Y., Courville, A.Deep Learning, 2016, Section 6.3.1 Rectified Linear Units and Their Generalizations
9. Cerda, P., Varoquaux, G., Encoding high-cardinality string categorical variables
10. Halford, M., Target Encoding Done The Right Way, link
11. Bengio, Y., Practical recommendations for gradient-based training of deep architectures, last revised 16 Sep 2012
12. Potdar, K., Pardawala, T., Pai, C. A Comparative Study of Categorical Variable Encoding Techniques for Neural Network Classifiers, October 2017.
13. Weinberger, K., Dasgupta, A., Langford, J., Smola, A., Attenberg, J., Feature Hashing for Large Scale Multitask Learning, Feb 2009
14. Jose, M., Fogliatto, F., Learning curve models and applications: Literature review and research directions 31 May 2011
15. Jiang, Y., Krishnan, D., Mobahi, H., Bengio, S., Predicting the Generalization Gap in Deep Networks with Margin Distributions, Google Research
16. Hawkins, D. M., The Problem of Overfitting, Oct 2003
17. Deng, L., Zhang, S., Balog, K., Table2Vec: NeuralWord and Entity Embeddings for Table Population and Retrieval, May 2019
18. Guo, C., Berkhahny, F., Entity Embeddings of Categorical Variables, April 2016
19. James, G., Witten, D., Hastie, T., Tibshirani, R., An Introduction to Statistical Learning 2017
20. [Arnold, B. C., Pareto Distributions, 2nd edition]
21. Guyon, I., A scaling law for the validation-set training-set size ratio
22. Guyon, I., Makhoul, J., Schwartz, R., Vapnik, V., What Size Test Set Gives Good Error Rate Estimates?

23. Mikolov, T. et al., Efficient Estimation of Word Representations in Vector Space
24. Erich Schikuta, E., Neural Networks and Database Systems, University of Vienna
25. van der Aalst, W. 2012. Process mining: Overview and opportunities. ACM Trans. Manage. Inf. Syst. 3, 2, Article 7 (July 2012),