universität wien

Bachelorarbeit

Benchmark for Recurrent vs. Deep Feedforward Neural Networks

Verfasserin ODER Verfasser

Fabio Schwinger

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2024

Studienkennzahl lt. Studienblatt:	A 033 521
Fachrichtung:	Informatik - Data Science
Betreuerin / Betreuer:	Dipl.-Ing. Dr.techn. Marian LUX

Contents

Abstract

This thesis describes the development from multiple simple JSON datasets to the evaluation and comparison of different feed forward and recurrent neural networks using Optuna for hyperparameter tuning.

The design decisions for each and every step during the workflow are explained in this thesis.

The recurrent neural network had simultaneously by far the best and also without a question the worst result, while the feed forward neural networks did not have this high variance in the performance.

Acknowledgement

I would like to express my deepest gratitude to my supervisor, Dipl.-Ing. Dr.techn. Marian Lux for his invaluable guidance and support throughout this project. His expertise has been instrumental in the completion of this thesis.

1 Motivation

Machine learning models predicting a user were never as accessible as in the last few years due to the progress in hardware, programming languages, libraries and the accessibility of big data on user behaviour. Modern companies attempt to get as much user data as possible to better understand the users' decision making process. Neural networks can help to capture subtle, unconscious decision patterns. Their ability to mimic the complexity of human brains makes them exceptional for this task, as the functionality of both neural networks and human brains can not be fully explained as of today.

The goal of this thesis is to build feed forward and recurrent neural networks for the given datasets and compare their ability to correctly classify the next page a user will access in the current session. As evaluation metric the accuracy is used, meaning the performance is evaluated by dividing the number of correctly predicted samples from an unknown test set by the total amount of predictions.

This thesis revolves particularly around the preprocessing of the data and the optimization of different hyperparameters.

The work process begun by taking a look at the datasets and to come up with an idea how to preprocess the data. In the next step I trained a simple, with the help of Keras Tuner optimized, neural network that still had a very broad range of possible hyperparamters. These results were then briefly assessed and afterwards I went back to try different other preprocessing ideas and evaluated them against each other. I started with the feed forward neural networks before working on the recurrent neural networks.

After I got the first prototypes running I researched the individual hyperparameters to define smaller and more accurate ranges to get more optimal hyperparameters. During this process I also switched from the Keras Tuner library to Optuna, as it is has more advanced features like pruning, the Optuna dashboard, the ability to integrate databases and it can be used with more machine learning libraries besides Keras. Due to this change the project is more flexible going forward.

During each phase of development I got constructive and helpful feedback making my work a lot easier.

2 Related Work

This thesis mainly discusses two fields of related studies - The comparison between feed forward neural networks and recurrent neural networks and the optimization of hyperparameters.

2.1 Comparison of the neural network types

Feed forward and recurrent neural networks were already compared in the past. They were mostly compared on two type of tasks - Natural Language Processing tasks and the task to predict a future event.

Both [31] and [30], authored partially by the same researchers, conduct comparative studies on the performance of feed forward and recurrent neural networks in speech recognition. They both came to the conclusion that recurrent neural networks outperform feed forward neural networks for this task. Similar to the goal of this thesis, these papers also compare feed forward and recurrent neural networks, but they do it for a completely different task as speech recognition has nothing to do with the task of predicting future events.

[28] compares the accuracy of a feed forward neural network, a recurrent neural network and an ensemble neural network predicting if an e-mail is a phishing e-mail or not based on 18 different binary encoded input features. The neural networks have also undergone some hyperparameter tuning for the learning rate and the number of hidden neurons. The final result for the best run and the averaged performance considering all different hyperparameters were quite similar, with a very negligible difference.

2.2 Optimizing hyperparameters

I found multiple papers describing the optimization for single hyperparameters like [2, 5, 7, 13, 16, 17, 29], but I did not find a scientific source focusing primarily on the optimization of multiple hyperparameters at the same time in a practical neural network. But I suppose this can be expected, since the best way to find the most optimal hyperparameters is to find them automatically using a library like Optuna. The automated process can be accelerated by defining a small interval for each different hyperparameter, so there are less different combinations to test.

This thesis can give a starting point regarding the optimization of multiple, different hyperparameters.

3 System Architecture and Design

This chapter defines and explains the key concepts and terminology relevant to neural networks. It includes an overview of feed forward neural networks (FFNN) and recurrent neural networks (RNN), describing their structures and differences. Additionally, the various hyperparameters used in the models will be explained, along with the reasoning behind the selection of these hyperparameters and their values or intervals.

3.1 What are neural networks?

A neural network is a model used in machine learning to make decisions in a manner that mimics the way the human brain makes decisions [15]. A neural network consists of at least one input layer and one output layer. Between these two layers, there can be an arbitrary number of hidden layers. Each layer consists of one or more neurons, also called units. The basic structure of a neuron is shown in Figure 1.
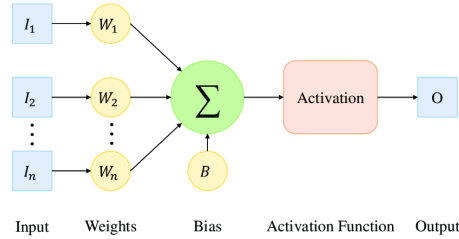
Figure 1: The structure of a neuron in a neural network [17]

3.2 Feed forward neural networks

Feed forward neural networks are the most basic neural networks. The other types of neural networks, are recurrent and convolutional neural networks. Feed forward models only use forward propagation, this means that the input data is only passed forward from layer to layer to make a prediction in the output layer at the end. Figure 2 shows a simple neural network with one hidden layer.
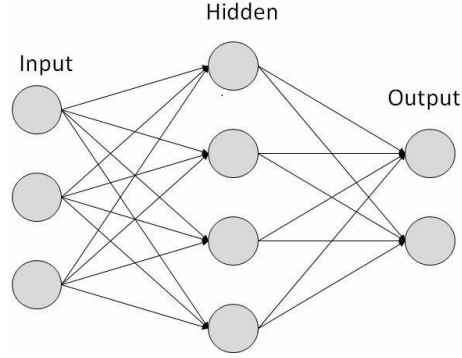
Figure 2: A simple feed forward neural network [20]

3.3   Recurrent neural networks

Recurrent neural networks are the other type of neural networks relevant for this thesis. In this type of neural networks, there are cycles within the hidden layers. These cycles allow us to keep previous input in mind and not only the current one [25]. Figure 3 shows a recurrent neural network with two hidden layers with a simple representation of the aforementioned cycles.
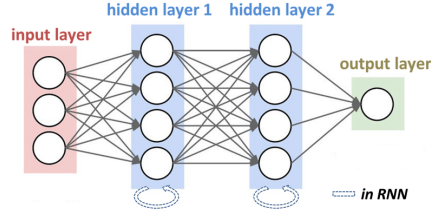
Figure 3: A simple recurrent neural network [19]

3.4   Differences between FFNN and RNN

Figure 2 and Figure 3 do a great job in showing the difference between feed forward and recurrent neural networks. Feed forward neural networks are only able to consider the current input and can not retain any information about previous inputs while recurrent neural networks can consider the current input and also multiple previous ones through the cycles inside the hidden layers.

3.5 Types of Variables

The input features can be split into different data types. Figure 4 shows the different types a variable can be split into.
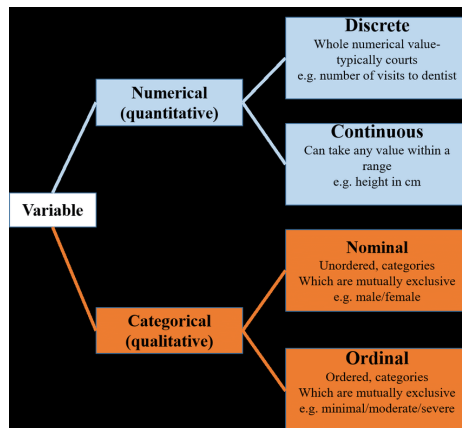
Figure 4: An image showing the different variable types
[21]

First we distinguish between categorical and numerical values. Examples for numerical values from our dataset include most weather variables like temp_max_c, wind_speed_kmh and prec_snow_mm_h. Numerical values do not need to encoded in comparison to categorical variables. Instead, they should be scaled so they all have the same scaling and order of magnitude. This ensures that no single numerical feature dominates the others, based on the original scaling. Numerical values can be further split into two subtypes: continuous and discrete. Figure 4 describes the difference between continuous and discrete variables.

Categorical values, on the other hand, do not need to be scaled down, but they need to be transformed into numerical values. Categorical values can also be further split into nominal and ordinal ones. The difference is that ordinal values have an inherent order, while nominal values do not.

3.6 Data Encoding

Data encoding is a crucial preprocessing step for machine learning models. Machine learning models only accept numerical input values and as a consequence all categorical variables need to be transformed into numerical values before using them to train a model. This can be achieved through different encoding techniques [23]. In the following segment, I will explain how the different encoding techniques, that were at least considered at some point, work.

3.6.1 One-hot Encoding

One-hot encoding creates a new column for every unique category in the original column. The newly added columns only contain binary values. Each row has one 1 value to represent the present original category and 0's elsewhere.

Figure 5: A simple table showing one-hot encoding

As we can see in Figure 5, we can transform the string values for each of the four colors into a unique numerical vector for each color through one-hot encoding. The main drawback of one-hot encoding is that it transforms one column of our dataset into n columns, where n is the amount of unique categories in this column. This can add a lot of new dimensions to our dataset. One-hot encoding works best for categorical nominal values with low to moderate cardinality, as one-hot encoding does not imply an order. The one-hot encoded value for blue $(1, 0, 0, 0)$ is not better or worse than the one-hot encoded value for the other colors. For high cardinality categories one-hot encoding is not advisable to use since it would highly augment the dimension of the dataset and fill each of the newly added dimensions with an immense amount of zeros.

3.6.2 Ordinal and label encoding

Ordinal and label encoding are very simple encoding techniques. They both work the same way, the only difference between these two is that label encoding is used for the column containing the labels, while the ordinal encoder is used for the input feature columns. These two encoders simply assign an integer value to each category. This does not add any new columns but implies an order for the feature which might not exist in reality [23]. The distance in this newly encoded data does also matter. For example in 6 blue (encoded as 1), is closer to green (encoded as 2) than to red (encoded as 3). Figure 6 shows a simple example how these encoders work.

Figure 6: A simple example showing ordinary encoding

3.6.3 Binary Encoding

Binary encoding is a combination of hash encoding and one-hot encoding [21]. It does not add as many new dimensions to the dataset as one-hot encoding and therefore binary encoding is more memory efficient compared to one-hot encoding.

First the categories are encoded using an ordinal encoding. These integer values are then transformed to binary values. Each digit from the binary value then gets split into its own column. [23]

I did not use binary encoding for any categorical value in the end due to the fact that it performed worse than one-hot encoding even for the higher cardinality features in the datasets. My observations regarding the performance of binary encoding align with those found in a thesis written by Cedric Seger (2018), who compared one-hot encoding, binary encoding, and feature hashing across two different datasets [26]. Figure 7 illustrates how the binary encoding works based on a simple example.
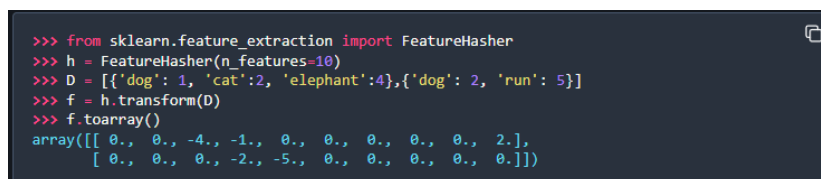
Figure 7: A simple example showing binary encoding

3.6.4 Feature Hashing

Feature hashing is a encoding technique to efficiently transform categorical features into numerical ones. This transformation works by mapping the unique features into an n-dimensional numerical space, where n is a hyperparameter that can be chosen freely. Large values for n result in high dimensional data, while a small n can result in hash collision, which means that different features have the same hash value. [22]

```
>>> from sklearn.feature_extraction import FeatureHasher
>>> h = FeatureHasher(n_features=10)
>>> D = [{'dog': 1, 'cat':2, 'elephant':4},{'dog': 2, 'run': 5}]
>>> f = h.transform(D)
>>> f.toarray()
array([[ 0.,  0., -4., -1.,  0.,  0.,  0.,  0.,  0.,  2.],
       [ 0.,  0.,  0., -2., -5.,  0.,  0.,  0.,  0.,  0.]])
```

Figure 8: Feature Hashing example by scikit-learn [22]

The Figure 8 shows a simple example of feature hashing in python where two dictionaries containing 2 respectively 3 features are transformed into a 10-dimensional feature vector.

I tested feature hashing for the features the most unique categorical values in the dataset, but I came to similar conclusion as with binary encoding. These features do not have enough unique values to make another encoding technique work better than one-hot encoding.

3.7 Embedding layers

Embedding layers are an alternative to one-hot encoding, due to the problem that one-hot encoding can get very impractical by adding a lot new dimensions. [14]

3.7.1 Input dimension

The input dimensions should be the same as the number of different unique categories you are encoding [14]. For example the column session_id from one of our dataset has 53 unique values and therefore the input dimension for the session_id embedding layer is 53.

3.7.2   Output dimension

The output dimension defines the dimension of the output vector from the embedding layer. For example regarding the session_id mentioned earlier. The input layer will have 53 different neurons, one for each unique category. The output layer will have a dimension of y, where y is the previously defined output dimension. Our 53 different unique categorical input values are now encoded into 53 unique output variables of length y. Since the output dimension is a hyperparameter there is no single "correct" value. The best value is somewhere between 1 and the input dimension. The smaller the value is the more information from the variable gets lost, while higher values result in a higher dimension and less compression. If the output dimension is the same as the input dimension the embedding works similar to one-hot encoding, because we get same dimensionality as if we one-hot encoded the variable in the first place.

3.8   Data Preprocessing

Real world data is likely to be incomplete, inconsistent, noisy and missing. [4]. Data pre-processing gets applied to this dirty real world data to get better quality data which results in better neural networks. In this thesis data pre-processing comprises of all steps taken up until the data is fed into the neural networks. The dataset, the different pre-processing steps and problems encountered with the dataset are part of this chapter. More general information about data pre-processing and the different pre-processing methods can be found in the paper by Famili et al. [12]

3.8.1   Dataset

I worked with multiple different JSON datasets. They contain some anonymised real world tracking data from users and also some weather data for the current day and the current hour for the tracked data.

The records in the different datasets are structured like this:

```json
"traces": [
    [
        {
            "time_dow": 0,
            "time_hod": 0,
            "time_utc": "2000-01-01T00:00:00.000Z",
            "device_id": "00000000-0000-0000-0000-000000000000",
            "content_id": "00000000-0000-0000-0000-000000000000",
            "event_data": {"Additional data that can be fetched via the event_type"},
            "event_type": 0,
            "oha_layout": 0,
            "session_id": "00000000-0000-0000-0000-000000000000",
            "time_local": "2000-01-01T00:00:00.000",
            "device_class": 0,
            "device_online": false,
            "content_portal": "string",
            "weather_day_id": 1234567,
            "device_platform": 0,
            "device_width_px": 0,
            "weather_hour_id": 123456789,
            "device_height_px": 0,
            "oha_language_iso2": 0,
            "device_orientation": 0,
            "device_country_iso2": 0,
            "device_language_iso2": 0,
            "weather_future_day_id": null,
            "weather_future_hour_id": null
        }
    ]
]
```

Via the id in weather_day_id and weather_hour_id it is possible to fetch more detailed weather data. The more in-depth data for the day look like this:

```
"weather_day_map": {
    "123456789": {
        "id": 123456789,
        "sun_set": "2000-01-01T00:00:00+00:00",
        "moon_set": "2000-01-01T00:00:00+00:00",
        "sun_rise": "2000-01-01T00:00:00+00:00",
        "moon_rise": "2000-01-01T00:00:00+00:00",
        "altitude_m": 0,
        "created_at": "2000-01-01T00:00:00.00000+00:00",
        "moon_phase": 0,
        "sunshine_h": 0,
        "temp_max_c": 0.1,
        "temp_min_c": 0.1,
        "geoposition": "0000000000000000000000000000000000000000000000000000",
        "calculated_at": "2000-01-01T00:00:00+00:00",
        "forecast_date": "2000-01-01",
        "prec_prob_pct": 0,
        "wind_strength": 0,
        "prec_rain_mm_h": 0,
        "prec_snow_mm_h": 0,
        "wind_direction": 0,
        "wind_speed_kmh": 0,
        "prec_total_mm_h": 0,
        "temp_felt_max_c": 0.1,
        "temp_felt_min_c": 0.1,
        "humidity_mean_pct": 0,
        "thunderstorm_prob": 0,
        "wind_speed_max_kmh": 0,
        "cloud_cover_max_pct": 0,
        "cloud_cover_min_pct": 0,
        "forecast_distance_h": 0,
        "cloud_cover_mean_pct": 0
}
```

The data for a specific hour timestamp is structured like this:

```
"weather_hour_map": {
    "1234567": {
        "id": 1234567,
        "temp_c": -0.1,
        "altitude_m": 0,
        "created_at": "2000-01-01T00:00:00.00000+00:00",
        "sunshine_h": 0,
        "geoposition": "00000000000000000000000000000000000000000000000000000",
        "temp_felt_c": 0.1,
        "humidity_pct": 0,
        "resolution_h": 0,
        "calculated_at": "2000-01-01T00:00:00+00:00",
        "forecast_time": "2000-01-01T00:00:00+00:00",
        "prec_prob_pct": null,
        "wind_strength": "string",
        "prec_rain_mm_h": 0,
        "prec_snow_mm_h": 0,
        "wind_direction": "string",
        "wind_speed_kmh": 0,
        "cloud_cover_pct": 0,
        "prec_total_mm_h": 0,
        "thunderstorm_prob": "string",
        "forecast_distance_h": 0
}
```

These three JSON objects are combined into one input feature vector. Besides these objects, the datasets contain four more JSON objects, with the names "content_map", "trace_enums", "content_enums" and "weather_enums". The "content_map" object contains additional information about the different content_id's, while the other three objects contain key-value pairs to convert the string values into integers.

3.8.2 Feature Selection

During the feature selection process a lot of different fields are getting removed, since they do not contain any data at all or any relevant information that helps the neural network to learn certain patterns in the data later on. The following features are all dropped in both datasets during the feature selection process.

```
['weather_day_id moon_set', 'weather_day_id moon_rise', 'content_portal',
'device_online','time_utc', 'time_local', 'device_height_px', 'device_width_px',
'weather_day_id sun_set', 'weather_day_id sun_rise', 'weather_day_id moon_set',
'weather_day_id moon_rise', 'weather_day_id created_at', 'weather_day_id calculated_at',
'weather_day_id forecast_date', 'weather_hour_id created_at',
'weather_hour_id calculated_at', 'weather_hour_id forecast_time',
'weather_future_day_id', 'weather_future_hour_id', 'event_data.for_date']
```

The weather_future_hour_id and the weather_future_day_id are dropped because, they do not contain a value in any of the datasets.

The content_portal value is dropped because it contains only a single unique value and the network will not learn anything from this field if it always contains the same value.

The device_online field is dropped due to similar reasons, as it contains only boolean values and this value is true in almost all cases.

All the other features are dropped for the reason that they are all absolute values like exact timestamps. All these are not important when it comes to the training of the neural networks later on, because these exact values are unique for a single record and will never get reproduced again.

3.8.3   Feature Encoding of the remaining fields

All weather features that are not numerical values already need to get transformed to their numerical value that is already saved in the "weather_enum" object in the JSON file.

The cyclical fields need an additional pre-processing step that is explained in more detail later on. Afterwards they need to be scaled like any other of the ordinal fields. All the nominal feature are scaled using a standardization technique like scaling to ensure that they all have the same order of magnitude.

The remaining categorical features, except the different id's, are all encoded using a one-hot encoder as it results in the best performance like I explained earlier when presenting all the different categorical encoding techniques that I have considered.

3.8.4 Addition of the previous record and the record before that in a session

A specific requirement for the feed forward neural network was to include the previous two records of a session to enhance comparability with the recurrent models. A challenge emerged in handling cases where these previous records do not exist, such as with the first or second record of a session. The ideas I have considered are:

1. Add completely new records containing the data of the first existing record and add two new features to all records.

2. Add new records with the mode values of the complete session for each feature and add two new features to all records.

3. Only add two new features to existing records.

At the end I chose the last design. Rather than adding two records with the exact same data, except for a placeholder in the content_id field, of the first record or records with the mode values of a session, I decided to only add placeholders to the existing records if the previous records would not exist. This approach makes sure that the dataset does not get bloated with synthetic data that also distorts the information from the existing records.

3.8.5 Missing values

For the fields where only a small amount of data points were missing I substituted these missing values with the mode of this field. The mode is the value that appears the most often in the whole dataset.

3.8.6   Challenges with the Datasets

The main difficulty was the imbalance in the label field. As can be observed in the following Figure 9 very few content_id's appear very often in the dataset, while the majority of the content_id's only appear very few times.
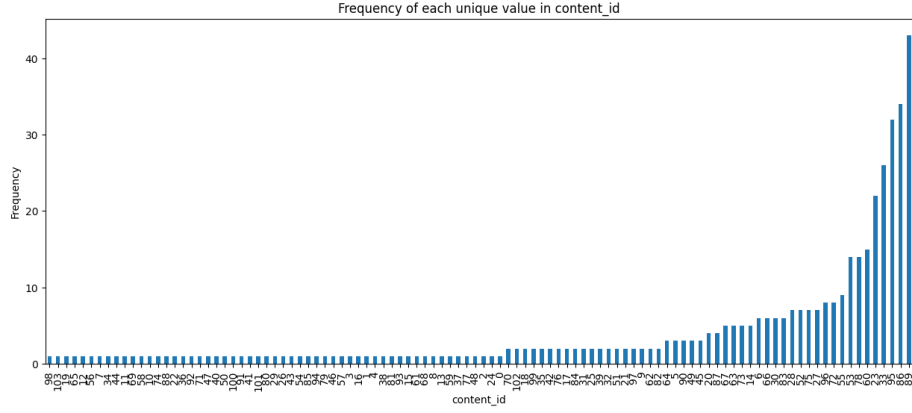
(a) The frequency of the different content_id's in the larger original dataset
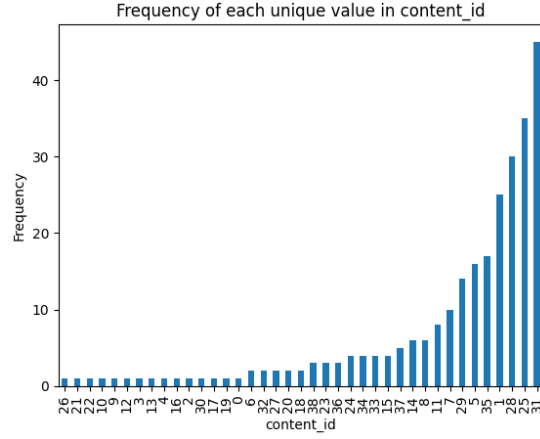
(b) The frequency of the different content_id's in the smaller original dataset

Figure 9: The frequency of the different content_id's in both datasets

The $n + 1$ content_id get used as label to evaluate the prediction made by the neural network. Synthetic Oversampling of the dataset does not work because most of the content_id's do not even appear often enough in the label set to generate new samples. Other oversampling techniques to balance the datasets like RandomOverSampling also do not work, due to the way how the RandomOverSampler works, as it takes a record from the minority classes and adds them an additional time until all the labels appear a balanced amount [18]. Adding the same single record multiple time does not make the neural network perform better, because it learns to associate this exact record with the label and as soon as only some features are changed even by a small margin, the neural network will not be able to link the different record to the label anymore.

A problem specific for the recurrent neural networks was coming up with a good sequence length. I tried the shortest and the longest session length as sequence length, the average session length and some other sequence lengths based on the shortest sequence (shortest sequence times two and some other comparable experiments). In the end I went with the median value, as it worked better or equally good for the sequence lengths I have tried, but it has the advantage that it is better usable for future datasets, as it is not prone to outliers like the smallest, longest or the average sequence length.

The preprocessing steps in general did pose a lot of questions that needed to be answered like "Which features to keep and why?", "How to encode the features?", "What to do with null values?", "How to proceed with records inside a session that don't have one or two previous records?", "Can I fix the unbalance of the dataset?" to name a few.

3.8.7 Removal the low appearing content_id's

Figure 9 displays the unbalance in the content_id field which is also used for the labels. To reduce the amount of unique labels and to improve the performance of the neural network a bit, I decided to remove the content_id's that appear only once.

3.8.8 Removal of the last record of every session

The label of each record is the $n + 1$ content_id, but since the content_id for the last record of a session does not exist, I remove each of the last records instead of adding a placeholder value as label. Each added placeholder value corrupts the information from the original data, by adding a synthetical value that does not contain any relevant, actual information.

3.8.9 Encoding of cyclical features

Features describing for example the time are cyclical. Cyclical features need an extra pre-processing step to ensure that the neural network correctly interprets them. If we leave the hour unencoded our model will correctly calculate the difference between 10 p.m. and 11 p.m. as 1, but it will fail to calculate the difference from 11 p.m. to 12 a.m. The difference in the second example would be -23 even though the time difference is the same as in the example before [1]. The same happens with all other cyclical features, when jumping from 59 to the next minute or second, from December to January when encoding months or when jumping from Sunday to Monday for weekdays for example.

However this problem can be fixed by splitting the feature into two new features and performing a sine transformation on one of the features and a cosine transformation on the other one. The formulas for these transformations are [1]:

$$x_{sin} = sin(\frac{2 \cdot \pi \cdot x}{max(x)})$$

$$x_{cos} = cos(\frac{2 \cdot \pi \cdot x}{max(x)})$$

with x being the value of the cyclical feature.

By applying this transformation the issue regarding cyclical features gets resolved, as the difference between the examples from before are now calculated correctly due to properties of the sine and cosine functions.

3.8.10 Train, test and validation split

The last step of data pre-processing was coming up with an appropriate split for the train, test and validation sets. To calculate the best split I took inspiration from this paper by Roshan [16], which comes up with a mathematical formula to get the optimal split size.

After the feature selection step, the original datasets have 65 respectively 64 features. Applying the mathematical formulas by the paper mentioned beforehand, the optimal split ratio can be calculated with the following formula: $\delta^* = \frac{1}{\sqrt{(p)}+1}$ with p being the number of features. The training, validation and test sets should be split according to this ratio $p : \sqrt{(p)} : (\sqrt{(p)} + 1)$.

Now we calculate δ^* with $p = 65$, which gives us the result $\frac{1}{\sqrt{(65)}+1} \approx 0.11$.
With δ^* we can now calculate the individual split sizes:

Training set:
$$(1 - \delta^*)^2 = (1 - 0.11)^2 \approx 0.79$$

Validation set:
$$\delta^*(1 - \delta^*) = 0.11(1 - 0.11) \approx 0.10$$

Test set:
$$\delta^* \approx 0.11$$

The calculations for the other dataset stay approximately the same as they only differ by one feature. For simplicity reasons I decided to use a 80/10/10 instead of a 79/10/11 split.

3.8.11   Sequence length for the recurrent neural networks

Figure 10 shows all unique sequence lengths and how often they appear in the data set after removing the sessions with sequence with length 1. The last record is already cut in each of the sequences, so the sequences with length 1 originally had a length of 2 and so on.
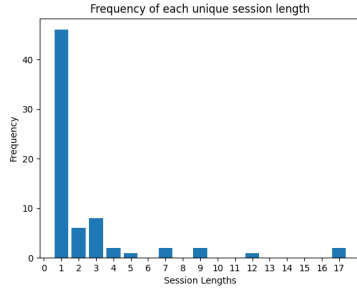
(a) The sequence lengths of the smaller original dataset
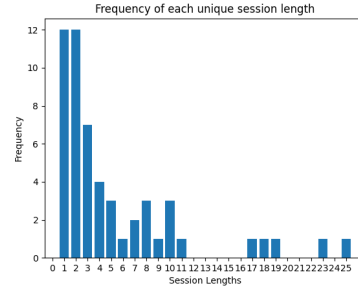
(b) The sequence lengths of the larger original dataset

Figure 10: The sequence lengths of both datasets

3.9   Padding

Recurrent neural networks take sequential inputs of equal length. Therefore we need to pad all inputs to the same length. For padding techniques we can differentiate between padding and truncation. [11]

There are two different possible methods regarding the truncation or the padding of the input. [11]

For truncating pre-sequence or post-sequence truncation is available. Presequence truncation signifies that we shorten a too long sequence by removing values at the beginning of our sequence. Post-sequence truncation on the other hand describes the effort to shorten the sequence by removing the values from the end of the sequence.

Padding has with pre-padding and post-padding the same two ideas as truncation. But instead of truncating the input, padding adds one or more placeholder value to beginning or the end of the sequence until we achieve the desired sequence length.

For my models I used a mix of post-sequence truncation and pre-padding. I tested different combinations and came to the result that this approach worked the best due to the fact that most of the sequences 10 are rather small and need to be padded to achieve the average sequence length. From my experience, it is better to add the placeholder values at the beginning than at the end. Only some sequences need to truncated and there it did not many any observable difference, if I truncate the beginning or the end of the sequence.

3.10   Cross Validation

Cross validation is a data resampling method to evaluate the ability of a supervised predictive model to correctly classify new unseen data and to prevent overfitting [9]. Cross Validation works by using different subsets of the original training set for training and testing. Figure 11 shows the theory behind five fold cross validation.
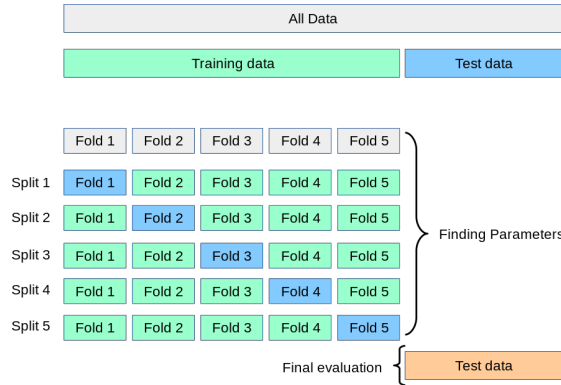

Figure 11: The theory behind five fold cross validation [22]

First the complete dataset gets split into a training and a test data. During the training, the training set gets split into k individual folds of the same size. $k - 1$ folds will be used as training data and the remaining fold will gets used as testing fold. This step is repeated k times. Each fold is being used exactly once as the testing fold and the remaining $k - 1$ times as training fold. After evaluating the model on each of the k testing fold, the average performance is computed to compare the different splits with each other. Afterwards the model is trained on the entire training set and evaluated on the held-out test set, split in the beginning from the complete dataset.

Based on the results reported by [24] and [9], I implemented ten-fold stratified cross-validation, as this strategy is suggested to achieve a good performance.

A lower amount of folds increases the variance of the model performance while a larger amount increases the bias in the model performance [24]. With a lower k value for the folds the variance increases since the individual folds get larger and more similar to each other, while the training sets also get smaller due to the test set getting larger, which also can lead to overfitting. While a larger value for k on the other hand might introduce a higher bias, which means that our model could get too simple and does not capture the underlying patterns correctly and underfits as a consequence, since it is harder to capture more complex pattern on smaller individual folds.

3.11 Model-Specific hyperparameters

In the following sub chapter I will explain how I decided on the different model specific hyperparameter and how I decided the minimum and maximum values and step sizes for them.

3.11.1 Number of Hidden Layers

The result from my literature review was that most neural networks will have two or less hidden layers. [13]. Therefore I decided to start with zero hidden layers, gradually increasing the number by one and evaluating the number of hidden layers that were chosen for the best performing model.

3.11.2 Units

While the number of units in the input and output layers are fixed, the number of units inside the hidden layers is an important hyperparameter, that also requires tuning. Having too few neurons in the hidden layers can lead to underfitting, meaning the model fails to adequately recognize patterns in the data, resulting in poor performance. Too many neurons on the other hand can result in overfitting, describing the phenomenon, that the model works well on training data but fails to perform on new unknown data. A large amount of neurons also increases the time needed to train the neural network. [13]

There is no particular rule defining the best amount of neurons for a hidden layer, but there are some guidelines to find the best amount of neurons for a specific model.

- The number of hidden neurons should be between the size of the input layer and the size of the output layer

- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.

- The number of hidden neurons should be less than twice the size of the input layer. [13]

These guidelines were used to define the lower and upper bound of neurons.

3.11.3 Activation functions

Activation functions are used to transform the input to an output, which is then fed into the next layer. These outputs get calculated with the input values, the weights of each neuron and the chosen activation function. Without these activation functions neural networks would work like linear regression models and lose a lot of performance and power in most cases. Many different types of activation functions exist. There is no definitive rule how to choose the correct activation function [27]. The activation functions gets automatically defined during the hyperoptimization process.

3.11.4 Loss Function

As loss function I have chosen the SparseCategorcalCrossentropy class, as this is the only probabilistic loss function designed for two or more as integer encoded labels in a classification task. [8]

3.11.5 Learning Rate

The learning rate is the most important hyperparameter. The optimal learning rate lies somewhere between 0.1 and 0.00001 according to the book by Bengio et al. [3]. Therefore I have chosen this interval as starting point before letting the hyperparameter tuner looking for the best value.
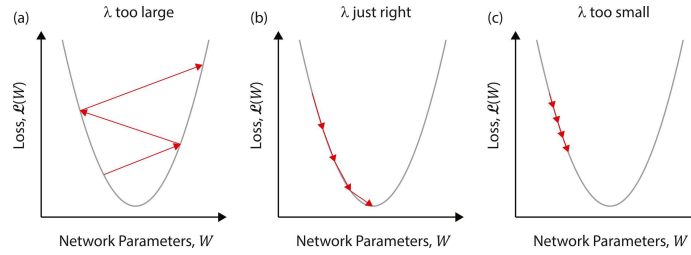
Figure 12: Showing the importance of the learning rate [10]

Figure 12 showcases the importance of finding the best learning rate. If the learning rate is too high the minimal loss gets overshot and we either oscillate between two similar losses or the loss even gets worse with every iteration like displayed in the Figure 12. If the learning rate is too small, the convergence towards the minimum is very slow and a lot of iterations are needed to achieve the best possible loss.

3.11.6 Optimizers

Optimizers are used to update the weights of all units after each epoch based on the chosen optimizer and the learning rate to minimize the predefined loss function.

There is no theory how to correctly chose an optimizer and the decision relies more on empirical evidence and benchmarking [7]. Therefore I try most of the different Keras optimizers during the search for the best hyperparameters. The Ftrl, Adamax and Adafactor optimziers do not get tested, since they are most suitable for different tasks. Ftlr works best with sparse datasets, while Adamax is suited to learn time-variant processes, like e.g. speech data and Adafactor is advised for Natural Language Processing tasks. [8]

3.11.7 Batch Normalization

Batch Normalization involves normalizing the output of a hidden layer before propagating it to the next layer. This technique improves the training of a neural network by allowing the use of higher learning rates without the risk of exploding gradients [5]. After each epoch, the weights of the units in each layer are updated. Without normalization, higher learning rates can cause some units' weights to "explode," leading to an unstable model.

3.11.8   Dropout and Dropout rate

Dropout is a technique used to combat overfitting in neural networks. Dropout helps to prevent overfitting by "dropping out" single units in the hidden and visible layers during training. The units which are dropped are randomly chosen. The probability of a unit getting dropped is described by the hyperparameter "dropout rate". The optimal retention rate is somewhere between 50% and 100% [29]. Figure 13 displays a simple example how dropout works.

(a) Standard Neural Net   (b) After applying dropout.
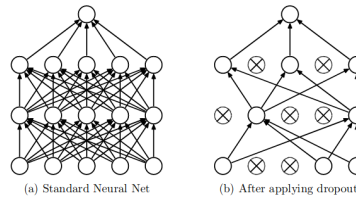
Figure 13: Simple example displaying how Dropout works

3.11.9   Batch Size

The batch size is a hyperparameter that describes the amount of samples that get used in one training step during neural network training. This hyperparameter does mainly impact training time and memory efficiency, as smaller batch sizes require less memory, due to the fact that less samples need to be loaded. [2]. The batch size gets automatically optimized during the hypertraining.

3.11.10   Epochs

The neural network processes the entire training set once per epoch. Thus, the number of epochs determines how many times this process is repeated. [6]. The hyperparameter number of epochs can be optimized with the help of the Keras EarlyStopping class. Using this class a metric, to monitor during the training process, can be defined and as soon as the selected metric stops improving for a defined amount of epochs the training process gets stopped and the remaining number of epochs do not get executed. [8]

4   Implementation

This chapter gives a detailed explanation about the core modules used, the implementation itself and how to run the project. As programming language I used python (3.12) as it is one of the best languages for machine learning tasks due to the additional libraries that exist. Additional libraries I used are pandas, numpy, scikit-learn, category_encoders and scipy during preprocessing.

Matplotlib is used to visualize the result, optuna to find the most optimal hyperparameter combination and keras along with tensorflow is used to implement the neural networks.

The project comprises of different main files, each running a different neural network model. Additionally, there are modules located in the Helpers directory, which contain shared methods utilized across all neural networks. Each main file is responsible for implementing specific steps in the overall process.

4.0.1 __init__

This file serves the purpose of marking the directory a package so that Python can import it into other files. It does not contain any executable code.

4.0.2 load_data

This module includes methods to read a JSON dataset and to convert it into a pandas DataFrame for subsequent preprocessing steps.

4.0.3 preprocess_dataframe

Here are methods to remove columns, fetch the weather data from the JSON, replace null values and encode all the different features to integers, so they can be fed into the neural networks later on. It also contains a method to move a specific column to the front of the dataset. This code is used to move the features that will later be embedded to the front, so they can be easier split into their own arrays after building the input feature vectors.

4.0.4 preprocess_feature_vectors

In this module methods to break the DataFrame into two different numpy arrays can be found. One of these arrays contains the input features and the other one the classification labels. There are also some methods to calculate the sequence length for the recurrent neural network and to pad the input features to the previously defined sequence length. Additionally it contains a method to calculate the input dimension needed for the embedding layers.

4.0.5 split_sets

The functionality to split the data for the embedding layers from the input feature vectors is defined in this module. There are two different methods, as the array containing the input features has a different dimension in the feed forward neural network compared to the recurrent neural network. The feed forward network does not specifically group the input feature vectors by their session and therefore the array that gets passed as parameter has one dimension less in contrast to the array for the recurrent neural network.

4.0.6 plot

This modules implements the methods to plot the data returned by the training process of the neural networks. It also contains a method to plot the unique values of a defined column in the DataFrame, this code was primarily used to explore the Dataset and to generate Figures used in this thesis.

4.0.7 model

This is a superclass that defines a constructor for all attributes and a method that calls the different plot methods after the the training.

4.0.8 ffnn_model

This subclass inherits the attributes and methods from the superclass and implements the different methods needed to build, hypertune and train the feed forward neural network. hypertune_model defines the optuna study to optimize the hyperparameters. The objective method defines the lower and upper boundary for the hyperparameters and the cross validation strategy. create_model is used to build the feed forward neural network with the different hyperparameters during optimization and later the method is used to build the model with the hypertuned parameters. train_model implements the code to train the model with the best parameters saved in an optuna.trial.FrozenTrial object.

4.0.9 rnn_model

This module contains similar methods with comparable functionality to the other subclass.

Both models are trained with a callback that stops the training if the loss stops improving for some epochs, therefore they are both called with a very high number of epochs. The training will finish in less epochs.

4.0.10 ffnn_only_content_id

This is the same class as ffnn_model except everything regarding the device_id is removed.

4.0.11 rnn_only_content_id

This is the same class as rnn_model except everything regarding the device_id is removed.

4.0.12 rnn_no_previous

This is the same class as rnn_model except everything regarding the prev_content_id and the prev_prev_content_id is removed.

4.1 How to run the project

The source code for this project is available on GitHub. The project can be
build like this:

```
$ git clone https://github.com/RogueRefiner/BachelorThesis.git
$ cd BachelorThesis
$ pip install -r requirements.txt
```

After all additional modules are installed with the help of the requirements.txt
file, one can execute the modules with the command:

```
$ python <file.py>. e.g. simpleRNN.py
```

5 Evaluation and Discussion

This chapter comprises lessons I learned while working on this thesis, the results
of the different types of neural networks on the two different datasets and a
discussion about the results.

5.1 Lessons learned during Implementation

This project did teach me a lot of lessons about the architecture of feed for-
ward neural networks and recurrent neural networks in an environment without
already cleaned datasets with a low amount of different features. Most of the
datasets I used in previous lectures or privately were already preprocessed to a
degree that there was not a lot data cleaning or hyperparameter tuning neces-
sary to achieve good results.

It did also educate me in how to use libraries like Optuna or keras tuner to
take care of hyperparameter tuning in an automatic fashion and how to define
good upper and lowerbounds for the different hyperparameters.

But mostly this project taught me how to clean a dataset. I already knew
previously how important it was to work with a good dataset when building
a neural network, but the work really showed me the impact of bad decisions
during the preprocessing phase.

5.2 Evaluation of the bugged models

The code used during the training of these models had a bug and as a consequence not the whole datasets were correctly read. Yet all of the models were trained with the same problem and are therefore still comparable.

To evaluate the models I hypertrained each model two different times and evaluated each of them two times. The performance of a model is mainly assessed by the accuracy on the test set. As this metric directly tells us the performance of our model on previously unseen samples.

Table 1 describes how many unique content and session id's and how many samples the different datasets had before the data preprocessing steps.

	# of content id's	# of sessions id's	# of samples
small_original	39	80	267
large_original	104	76	411
large	113	121	460
large no_search	8	121	460
medium	282	955	9298
medium no_search	23	955	9298
small	31	36	171
small no_search	3	36	171
hotel s	138	142	1108
hotel s no_search	10	142	1108
hotel w	110	132	1079
hotel w no_search	8	132	1079

Table 1: Summary of Samples and Unique Values in Different Columns Before Preprocessing

Table 2 shows how the number of unique content id's, session id's and samples evolved after the preprocessing steps. The table also contains the results for the feed forward neural networks and recurrent neural networks. Most of the times the results are very comparable. The recurrent neural networks only significantly outperforms the feed forward neural networks for two datasets, while the feed forward neural networks outperforms the recurrent with a notable difference four times. In this four cases the recurrent neural network performs very badly. This contrast is a result of the fact that the sessions are more important for the recurrent neural networks. The large dataset for example has 284 samples but only 68 different sessions. Therefore the recurrent neural network tries to learn 53 different content_id's with 61 training and 7 test samples, in the meantime the feed forward neural network uses 255 training and 29 test samples.

	# of content id's	# of sessions id's	# of samples	FFNN Accuracy	RNN Accuracy
small_original	24	79	173	51%	50%
large_original	49	74	284	25%	0%
large	53	118	284	26%	0%
large no_search	5	121	337	79%	38%
medium	215	955	8280	49%	16%
medium no_search	13	955	8331	82%	83%
small	24	36	127	37%	44%
small no_search	3	36	135	87%	100%
hotel s	86	142	947	51%	50%
hotel s no_search	8	142	995	85%	80%
hotel w	66	129	912	33%	45%
hotel w no_search	6	132	952	93%	91%

Table 2: Evolution of the parameters after preprocessing and the results of the networks

Table 3 shows the accuracy of the feed forward and the recurrent neural network when omitting the weather data for the original two datasets. The weather data does not influence the result of the feed forward neural networks. However the recurrent neural network has a much better performance without the weather data. The result for the larger dataset is as bad as with the weather data, but I would suspect that this has more to do with the dataset itself and not as much with the presence of the weather data.

	FFNN Accuracy	RNN Accuracy
small_original	49%	82%
large_original	25%	4%

Table 3: Results of the neural networks without weather data

Table 4 shows the performance of the recurrent neural networks without the previous content_id and the content_id before that. As we can see the result is practically the same as for the test without the weather data. Keeping either of them improves the performance a little bit as we can see in 5, but keeping both of them in the same model impairs the performance by a significant margin 2.

	Accuracy
small_original	82%
large_original	0%

Table 4: Results of the recurrent neural networks without the prev and the prev_prev_content_id

Table 5 displays the result of the feed forward and the recurrent neural network for the first two datasets when removing all features except the content_id, prev_content_id, prev_prev_content_id and the session_id. Here it is observable that most of the networks performed better than with all the different features, especially the recurrent neural networks improved a lot. The only network that performed worse was the feed forward neural network with the smaller dataset. The performance got dragged down by an outlier in the evaluation process, as can be seen the figure describing the test result in the GitHub repository: `https://github.com/RogueRefiner/BachelorThesis/blob/main/figures/results/test/re_ffn_smaller_original_only_content.png`. But even without this outlier the performance would have been worse, while the other feed forward neural network improved by approximately 12% and the recurrent neural networks by 25% each. This is also the first time the recurrent neural network performed for the larger original dataset, the performance is still bad, but it is better than 0% or 4% from all other tests on this dataset.

	FFNN Accuracy	RNN Accuracy
small_original	24%	75%
large_original	37%	25%

Table 5: Results of the neural networks with only content_id, prev_content_id, prev_prev_content_id and session_id as features

The recurrent network performed better in each special setup than with the originally chosen features, while the feed forward neural networks improves one time and the other time it stays the same.

5.3   Evaluation of the fixed models

In the following sections most of the neural models were trained an additional times after fixing the error from the previous subchapter. Due to hardware constraints not all models are trained again. All models were only trained a single time compared to the two times before.

As we can see when comparing this table of results to the table 2, the bugfix did not influence most of the datasets. The RNN for the large_no_search improved by a significant margin, whereas the RNN for the hotel_s dataset got a lot worse. For the small dataset both of the models had a notable improvement.

	# of content id's	# of sessions id's	# of samples	FFNN Accuracy	RNN Accuracy
small_original	37	220	535	59%	50%
large_original	830	10680	32241	-%	-%
large	544	6127	20031	9%	19%
large no_search	50	6151	20292	75%	57%
medium	293	2717	19735	44%	14%
medium no_search	18	2739	19845	81%	73%
small	42	566	959	59%	76%
small no_search	6	566	967	80%	80%
hotel s	373	4815	22902	41%	3%
hotel s no_search	23	4822	23023	85%	78%
hotel w	675	12371	51849	-%	-%
hotel w no_search	43	12388	51617	-%	-%

Table 6: Evolution of the parameters after preprocessing and the results of the networks

When removing the weather data both models for the smaller_original dataset were affected. The FFNN improved by a relevant margin, while the RNN got worse by a similar margin.

	FFNN Accuracy	RNN Accuracy
small_original	61%	65%
large_original	-%	-%

Table 7: Results of the neural networks without weather data

The RNN without the prev_prev and the prev_content_id was only tested for the smaller_original datasets again. In this instance the bugfix had no impact on the result.

	Accuracy
small_original	80%
large_original	-%

Table 8: Results of the recurrent neural networks without the prev and the prev_prev_content_id

The FFNN for the small_original datasets improved by 20% after the bugfix, while the performance of the RNN was exactly the same.

	FFNN Accuracy	RNN Accuracy
small_original	44%	75%
large_original	-%	-%

Table 9: Results of the neural networks with only content_id, prev_content_id, prev_prev_content_id and session_id as features

Overall it can be observed that the no_search variants of the datasets outperform their counterparts with more content_id's. When taking a look at the results of the datasets with all the remaining features after the preprocessing steps all of the no_search datasets, except for one, achieve a good result (Accuracy $>= 70\%$), whereas without the bugfix not a single dataset and after the error got resolved only a single of the other model surpassed this 70% threshold. Some of the datasets get reasonably close to the threshold, while other perform very very badly.

All figures describing the train and test results for the bugged and fixed models can be found in the GitHub Repository. Each bugged model got tested with 20 different hyperparameter combinations to make them comparable and to try to find a decent hyperparameter pairing, so none of the models performs very badly due to badly chosen hyperparameters, while the others got only tested with 10 different hyperparameter combinations to keep the training time in a reasonable time frame.

6 Future Work and Conclusions

This section delves into improvements and potential future implementations, concluding with a small, concise overview summarizing the project.

6.1 Future Work

There are still several aspects that can be further improved in the future:

1. The session length as integer could be included as additional input feature for the feed forward neural network. At the moment only the $n - 2$-th, $n - 1$-th and the $n + 1$-th record are relevant and considered for the n-th record in a session.

2. A Optuna Dashboard can be implemented to make the hyperparameter optimization visually more appealing.

3. Run a larger amount of trial and implement the calculation of intermediate values and the ability to prune them, if they are not promising enough. Optuna already provides interfaces to implement pruning.

4. Change the mode of the whole dataset to the mode of the specific session where a null value gets substituted.

5. Test the more specific parameter sets (no weather data; only content_id, prev_content_id and prev_prev_content_id; recurrent neural network without prev_content_id and prev_prev_content_id) for all the other datasets.

6.2 Conclusion

In conclusion, this thesis presents an open-source basis to preprocess JSON datasets, transform the data and feed it into a feed forward or a recurrent neural network. The implementation is based on python and multiple of its well-known libraries. Python is one of the most popular programming languages, especially in machine learning, therefore the code basis will not become obsolete in the near future and be taken as starting point for further improvement.

7 Appendix

7.1 Source Code

The source code, an in-depth explanation how to run the project with an example dataset, and all figures created for this thesis can be found on the git repository `https://github.com/RogueRefiner/BachelorThesis`

References

[1] Avan Wyk, J. Encoding cyclical features for deep learning, 2022.

[2] Bengio, Y. Practical recommendations for gradient-based training of deep architectures. *CoRR abs/1206.5533* (2012).

[3] Bengio, Y., Goodfellow, I., and Courville, A. *Deep learning*, vol. 1. MIT press Cambridge, MA, USA, 2017.

[4] Bhaya, W. Review of data preprocessing techniques in data mining. *Journal of Engineering and Applied Sciences 12* (09 2017), 4102–4107.

[5] Bjorck, N., Gomes, C. P., Selman, B., and Weinberger, K. Q. Understanding batch normalization. *Advances in neural information processing systems 31* (2018).

[6] Brownlee, J. What is the difference between a batch and an epoch in a neural network. *Machine learning mastery 20* (2018).

[7] Choi, D., Shallue, C. J., Nado, Z., Lee, J., Maddison, C. J., and Dahl, G. E. On empirical comparisons of optimizers for deep learning, 2020.

[8] CHOLLET, F., ET AL. Keras. `https://keras.io/2.15/api/layers/core_layers/embedding/`, 2015.

[9] DANIEL, B. Cross-validation.

[10] DENG, Y., REN, S., MALOF, J., AND PADILLA, W. J. Deep inverse photonic design: A tutorial. *Photonics and Nanostructures - Fundamentals and Applications 52* (2022), 101070.

[11] DWARAMPUDI, M., AND REDDY, N. V. S. Effects of padding on lstms and cnns. *CoRR abs/1903.07288* (2019).

[12] FAMILI, A., SHEN, W.-M., WEBER, R., AND SIMOUDIS, E. Data preprocessing and intelligent data analysis. *Intelligent Data Analysis 1*, 1 (1997), 3–23.

[13] HEATON, J. *Artificial Intelligence for Humans: Deep learning and neural networks.* Artificial Intelligence for Humans Series. Heaton Research, Incorporated., 2015.

[14] HEATON, J. What are embedding layers in keras (11.5). `https://www.youtube.com/watch?v=OuNH5kT-aD0`, 2019.

[15] IBM. What is a neural network? `https://www.ibm.com/topics/neural-networks`, n.d.

[16] JOSEPH, V. R. Optimal ratio for data splitting. *Statistical Analysis and Data Mining: The ASA Data Science Journal 15* (04 2022).

[17] KONG, F., WANG, X., PU, K., ZHANG, J., AND DANG, H. A practical non-profiled deep-learning-based power analysis with hybrid-supervised neural networks.

[18] LEMAÎTRE, G., NOGUEIRA, F., AND ARIDAS, C. K. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research 18*, 17 (2017), 1–5.

[19] MA, S., XIAO, B., HONG, R., ADDISSIE, B. D., DRIKAS, Z. B., ANTONSEN, T. M., OTT, E., AND ANLAGE, S. M. Classification and prediction of wave chaotic systems with machine learning techniques.

[20] MA, X. Initial margin simulation with deep learning.

[21] MWAMBA, D., AND JOE, I. A deep-learned embedding technique for categorical features encoding.

[22] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PASSOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND DUCHESNAY, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[23] POTDAR, K., PARDAWALA, T., AND PAI, C. A comparative study of categorical variable encoding techniques for neural network classifiers.

[24] RON, K. A study of cross-validation and bootstrap for accuracy estimation and model selection.

[25] SCHMIDT, R. M. Recurrent neural networks (rnns): A gentle introduction and overview.

[26] SEGER, C. An investigation of categorical variable encoding techniques in machine learning: binary versus one-hot and feature hashing, 2018.

[27] SHARMA, S., SHARMA, S., AND ATHAIYA, A. Activation functions in neural networks. *Towards Data Sci 6*, 12 (2017), 310–316.

[28] SOON, G. K., ON, C. K., RUSLI, N. M., FUN, T. S., ALFRED, R., AND GUAN, T. T. Comparison of simple feedforward neural network, recurrent neural network and ensemble neural networks in phishing detection. *Journal of Physics: Conference Series 1502*, 1 (mar 2020), 012033.

[29] SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research 15*, 1 (2014), 1929–1958.

[30] SUNDERMEYER, M., NEY, H., AND SCHLÜTER, R. From feedforward to recurrent lstm neural networks for language modeling. *IEEE/ACM Transactions on Audio, Speech, and Language Processing 23*, 3 (2015), 517–529.

[31] SUNDERMEYER, M., OPARIN, I., GAUVAIN, J.-L., FREIBERG, B., SCHLÜTER, R., AND NEY, H. Comparison of feedforward and recurrent neural network language models. In *2013 IEEE international conference on acoustics, speech and signal processing* (2013), IEEE, pp. 8430–8434.