Applying deep neural networks in life insurance

Part 1 - Introduction and overview

Axel Helmert, Vienna, July 2020
msg life central europe gmbh

Deep Neural Networks (DNNs) in Life Insurance

- AI and machine learning are already reality in the insurance world.
- So far, however, there have been no productive applications of machine learning in the core life insurance business, for example the calculation of tariff premiums or reserves.
- Before we start, I would like to say something about the history of deep neural networks.

Deep Neural Networks

History

- Inventors have long dreamed of creating machines that think. This desire dates back to at least the time of ancient Greece (think of Daedalus and others).
- Deep learning has a long history and has gone by many names and changing popularity.
- Deep learning models have grown in size over time as computer infrastructure (both hardware and software) has improved.
- Deep learning has solved increasingly complicated applications with increasing accuracy over time.
- There have been three important waves of development of deep learning (using different names):
  - Cybernetics in the 1940s –1960s,
  - Connectionism or parallel distributed processing the 1980s –1990s
  - And the current development under the name deep learning beginning in 2006

Deep Neural Networks

Learning from neuroscience?

- Some of the earliest learning algorithms we recognize today were intended to be computational models of biological learning, that is, models of how learning happens or could happen in the brain.
- As a result, one of the names that deep learning has gone by is artificial neural networks (ANNs).
- The modern term “deep learning” goes beyond the neuroscientific perspective.
- It appeals to a more general principle of learning multiple levels of composition, which can be applied in machine learning frameworks that are not necessarily neurally inspired.
- The earliest predecessors of modern deep learning were simple linear models motivated from a neuroscientific perspective.
- The McCulloch-Pitts neuron (McCulloch and Pitts, 1943) was an early model of brain function. This linear model could recognize two different categories of inputs.
- Today, neuroscience is regarded as an important source of inspiration for deep learning, but it is no longer predominant.

DNNs in Life Insurance

Universal Approximation Theorem / Regression

- We consider neural networks as a **functional approximation.**
- The goal is to approximate some function f^*. A network defines a mapping y=f(x;θ) and learns the value of the parameters θ that result in the best function approximation.
- The **Universal Approximation Theorem** provides the theoretical basis for this.
- The theorem was first proved in 1989 for a NN with sigmoid activation functions
- and then in 1991 for NNs with arbitrary non-linear activation functions.
- It states that any continuous function on compact subsets of $\mathbb{R}^n$ can be approximated to an arbitrary degree of accuracy by a feedforward NN with at least one hidden layer with a finite number of units and a non-linear activation.

Applying DNNs in Life Insurance

Universal Approximation Theorem / Regression

- Obviously, there are many possibilities to use DNNs in life insurance
- Taking into account the business case and the general conditions migration seems to be a very interesting field
- Migration deals with knowledge that is depicted in old (source-)systems, learning this knowledge and transferring it to a new (target-)system.
- This is an ideal situation for supervised learning.

Introduction to DNNs

Neurons – Repeat what we have learned already

Neurons are mathematical functions that can be defined as follows:

$$y = f(\sum_{i=1}^{n} x_i w_i + b)$$

- y is the output of the neuron. It is a single value.

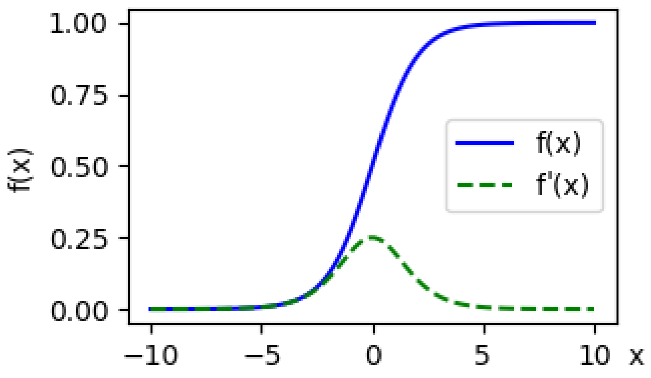
- *f* is a non-linear differentiable activation function. The activation function is the source of non-linearity in a NN—if the NN was entirely linear, it would only be able to approximate other linear functions.

- The argument of the activation function is the weighted sum (with weights w_i) of all the neuron inputs x_i (n total inputs) and the bias weight b. The inputs x_i can be either the data input values or outputs of other neurons.

Introduction to DNNs

Activation Functions

Sigmoid: Its output is bounded between 0 and 1 and can be interpreted stochastically as the probability of the neuron activating. Because of these properties, the sigmoid was the most popular activation function for a long time. However, it also has some less desirable properties (more on that later), which led to its decline in popularity. The following diagram shows the sigmoid formula, its derivative, and their graphs (the derivative will be useful when we discuss backpropagation):



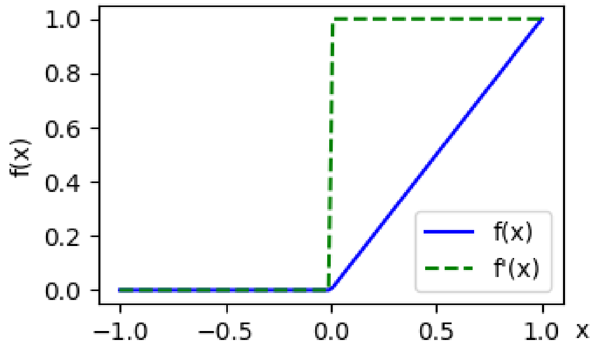
$$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$

$$f'(x) = \sigma(x)(1-\sigma(x))$$

Introduction to DNNs

Activation Functions

***LU:** family of functions (**LU** stands for **linear unit**). We'll start with the rectified linear unit (**ReLU**), which was first successfully used in 2011. The following diagram shows the ReLU formula, its derivative, and their graphs:

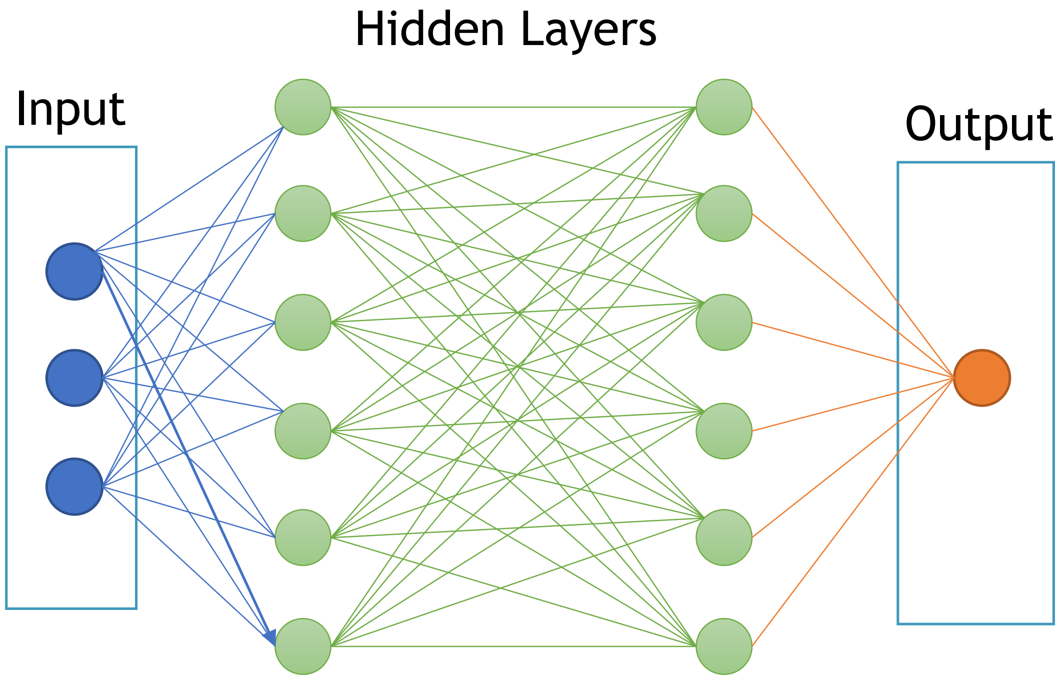


$$f(x) = \begin{cases} x \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$$

$$f'(x) = \begin{cases} 1 \text{ if } x > 0 \\ 0 \text{ if } x < 0 \end{cases}$$

DNNs in Life Insurance: Architecture

Supervised Learning: Regression with one unit in the Output-Layer



DNNs in Life Insurance: Quality and Trustworthiness

- For the training of DNNs, it is important to separate a training set from a test set (to control quality).
- The quality measures that are used are relevant.
- For life insurance values like tariff premiums or reserves, relative and absolute deviations are important.
- The measure can refer to a single test case, a set of test cases (a **batch**), or the entirety of the training data.
- The resulting distributions can also be specified.
- If the trained NNs are used productively in a target system, both the set of parameters for a call and the set of all possible call sets are finite, but too large to actually memorize them.

DNNs in Life Insurance: Quality and Trustworthiness

Quality is a measure for the deviations on the basis of the finite test data.

An $\varepsilon > 0$ is specified. The model is then tuned or (automatically) supplied with further training data until the desired quality is achieved (or the process stops).

Trustworthiness deals with statistical generalization.

It measures the probability $1 - \delta$ with which a predetermined quality is achieved in reality, that is with input data unknown during the training.

DNNs in Life Insurance: Performance and Technology

- The **performance** of trained DNNs calculating actuarial values is very good (in the millisecond range) and mostly as good as the performance of conventional programs.
- Which **technologies** do we use:
  - Python and jupyter notebooks (during training and documentation)
    - TensorFlow 2.x
    - Keras
    - Random Forest
  - Java (in productive environments)

DNNs at work: A demonstration

Training a DNN and calculate Tariff-Premiums

Method:

- Use of a specialized DNN: Supervised Learning
- trained with a policy adminstation system (here: msg.Life Factory)

Training Set: 100.000 calculated standard life insurance death benefit contracts

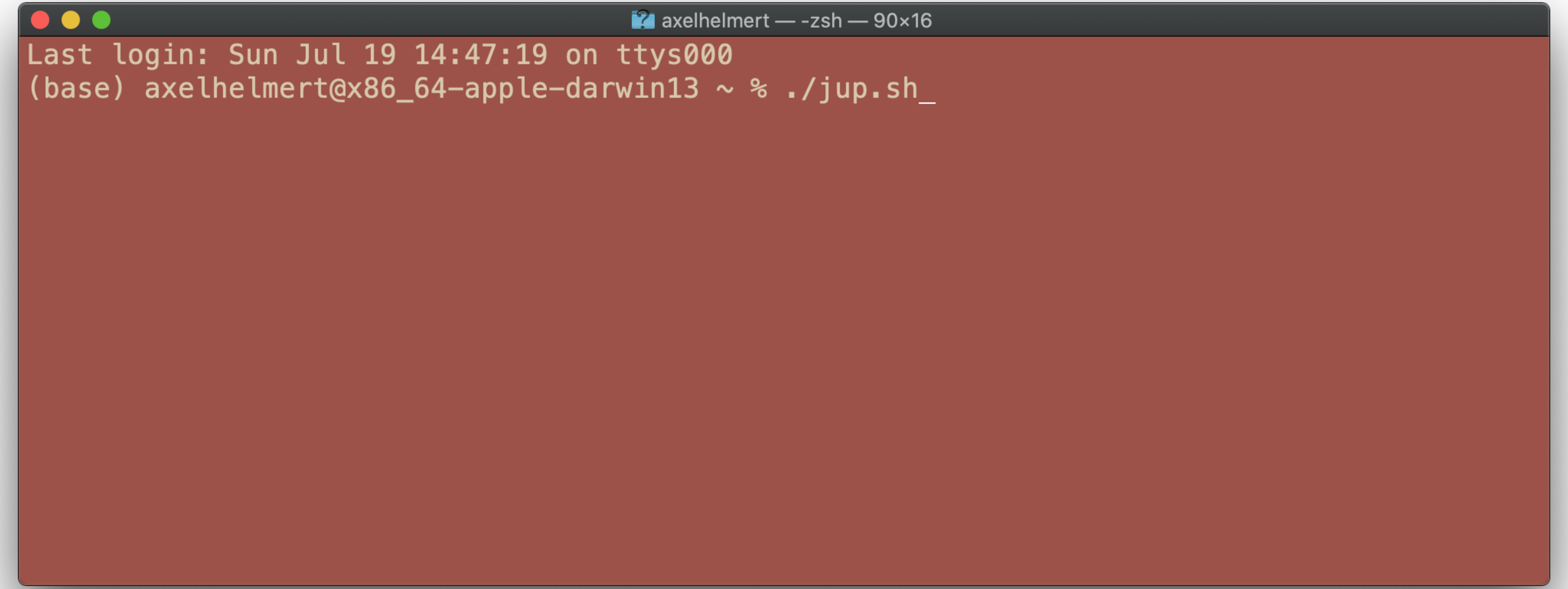
Benchmark: The tariff premium calculated by the DNN should not deviate more than 0.5% from the real contribution

Loss-Function: MSE (mean squared error).

The demo is run on a normal private computer.

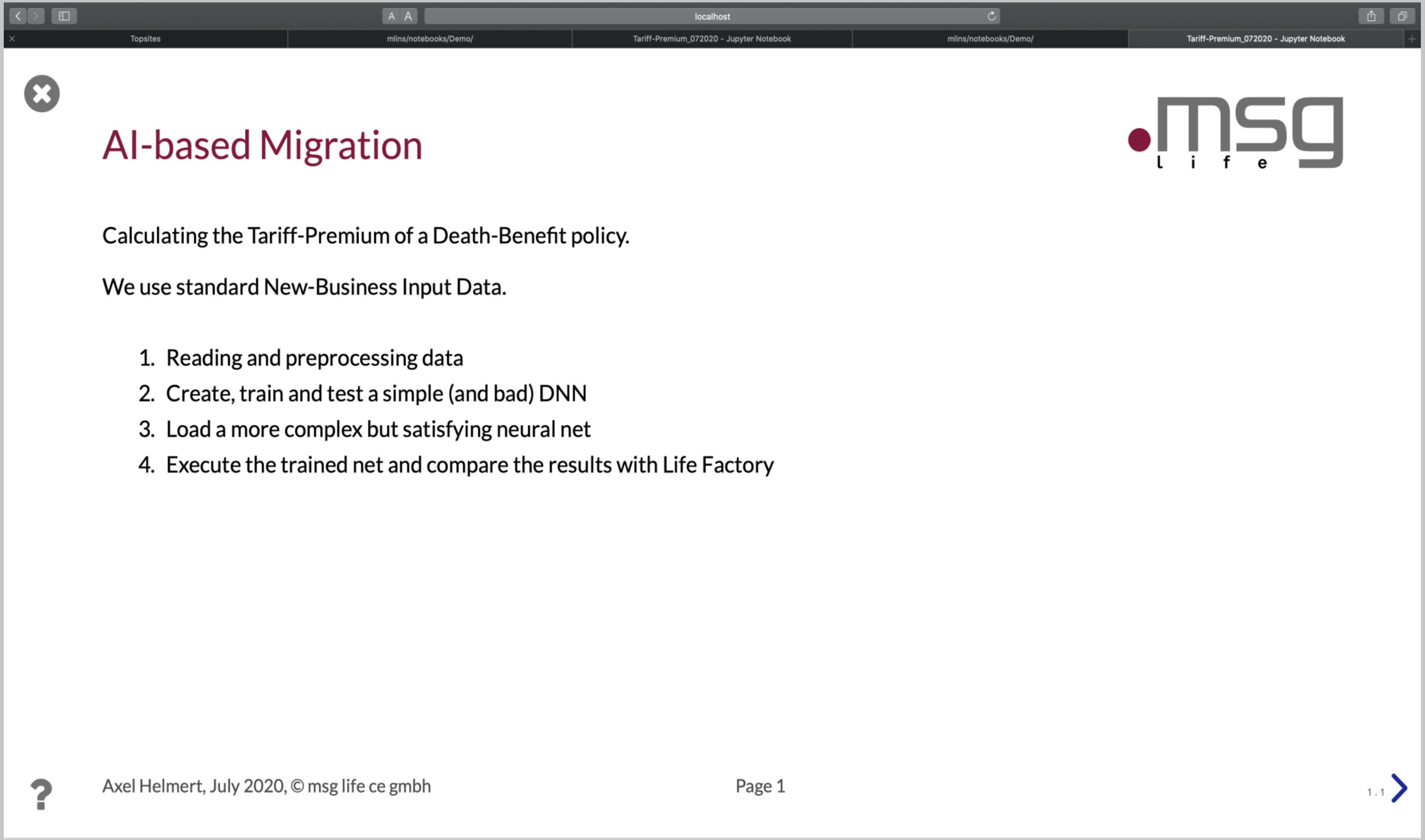
DNNs at work: A demonstration

I first open a terminal window. In the ML environment under Anaconda / Python I start a jupyter notebook that contains the application and some additional information (html / Tex).



DNNs at work: A demonstration

The first page / cell (here html) of the jupyter notebook with some explanations about the demo.



AI-based Migration

Read Data for Training and Testing.

In [1]:
```python
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd
from mlins.Tarife import preproc_Tarif_RI_2017

df = pd.read_csv("../../data/Tarife/Tarifierung_RI2017_large_Set.csv",sep=";",header='infer')
df.shape
```

Out[1]:
```
(100000, 10)
```

AI-based Migration

Show input data before preprocessing.

This is a typical input for a policy administration system.

In [2]:

```
df.head(4)
```

Out[2]:

	Beginnjahr	Beginnmonat	ZahlweiseInkasso	GeschlechtVP1	RauchertypVP1	x	n	t	Leistung	tba
0	2019	12	JAEHRLICH	WEIBLICH	RAUCHEN	55	4	1	234291.19	13893.98
1	2018	10	MONATLICH	WEIBLICH	NICHTRAUCHEN	53	13	10	365664.56	4995.25
2	2017	5	MONATLICH	MAENNLICH	NICHTRAUCHEN	50	2	1	754477.74	8581.78
3	2017	1	JAEHRLICH	WEIBLICH	RAUCHEN	29	20	6	354704.35	4481.04

Prepocessing (1/2)

In [3]:
```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, FunctionTransformer
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder
from sklearn.model_selection import train_test_split
```

Prepocessing (2/2)

In [4]:

```python
X_data = df.drop(["tba"], axis=1)
y_data = df["tba"]

cat_columns=['ZahlweiseInkasso', 'GeschlechtVP1','RauchertypVP1']
num_columns=list(X_data.drop(cat_columns,axis=1).columns)

X_data[num_columns]=X_data[num_columns].astype("float64")

scaler=StandardScaler()
num_pipeline=Pipeline([('scaler',scaler)])
cat_pipeline = Pipeline([('onehot',OneHotEncoder())])

data_preproc_pipeline = ColumnTransformer([
    ('cat_values',cat_pipeline,cat_columns),
    ('num_values',num_pipeline,num_columns)
    ])

X_train_raw, X_test_raw, y_train, y_test = train_test_split(X_data, y_data, test_size=0.2, random_state=1)

data_preproc_pipeline.fit(X_train_raw,y_train)
feature_names = list(data_preproc_pipeline.named_transformers_["cat_values"].named_steps["onehot"].get_feature_names()) \
+ data_preproc_pipeline.transformers_[1][2]

X_train = pd.DataFrame(data_preproc_pipeline.transform(X_train_raw),columns=feature_names)
X_test = pd.DataFrame(data_preproc_pipeline.transform(X_test_raw),columns=feature_names)
```

localhost

Topsites | mlins/notebooks/Demo/ | Tariff-Premium_072020 - Jupyter Notebook | mlins/notebooks/Demo/ | Tariff-Premium_072020 - Jupyter Notebook

Input-Data after Preprocessing:

In [5]:

```
X_train.head(6)
```

Out[5]:

	x0_HALBJAEHRLICH	x0_JAEHRLICH	x0_MONATLICH	x0_VIERTELJAEHRLICH	x1_MAENNLICH	x1_WEIBLICH	x2_NICHTRAUCHEN	x2_RAUCHEN	Beginnjahr	Beginnmonat	x	n	t	Leistung
0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.223756	1.592147	-0.723340	-0.190226	-0.477409	-1.395603
1	0.0	0.0	0.0	1.0	0.0	1.0	1.0	0.0	-0.002099	0.723194	-1.125855	-0.378880	0.341132	-0.026540
2	0.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	1.223756	0.433543	0.967220	-0.850515	-0.886680	1.219385
3	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.223756	0.723194	0.806214	-0.095899	-0.068138	1.387470
4	0.0	0.0	1.0	0.0	1.0	0.0	0.0	1.0	-0.002099	-1.014713	0.645208	-0.190226	-0.340986	-1.244856
5	1.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	-0.002099	-0.435411	-0.481832	0.187082	0.341132	-0.872374

?

Create Model (1/2)

In [6]:

```
from keras.models import Sequential
from keras.layers import Dense, Activation, advanced_activations, Dropout
from keras import backend as K
from keras import optimizers
from keras import losses

import mlins.metrics as M
import mlins.evaluation as evaluation
```

```
Using TensorFlow backend.
```

Create Model (2/2)

In [7]:

```python
def create_dnn(act_func_1='relu', act_func_2='linear', hidden_layer_size=40,
               hidden_layer_num=10, lr=0.0001, loss_func=M.K_mean_squared_relative_error,
               alpha=1.0, input_size=14, dropout_rate=0.0, dropout_rate_inp=0.0):

    model = Sequential()
    if act_func_1=='elu':
        act = advanced_activations.ELU(alpha=alpha)
    else:
        act = Activation(act_func_1)

     # first layer
    model.add(Dense(units=hidden_layer_size, input_shape=(input_size,)))
    model.add(act)
    if dropout_rate_inp: model.add(Dropout(dropout_rate_inp))
    # all other hidden layers
    for i in range(hidden_layer_num - 1):
        model.add(Dense(units=hidden_layer_size, activation=act_func_1))
        model.add(act)
        if dropout_rate: model.add(Dropout(dropout_rate))

    # final layer
    model.add(Dense(1, activation=act_func_2))

    model.compile(loss=loss_func,
                  optimizer=optimizers.Adam(lr=lr),
                  metrics=[losses.mean_squared_error, M.K_max_relative_error, M.K_relative_error_percentage, M.K_mean_squared_relative_error])

    return model
```

Model Training

In [8]:

```
model = create_dnn()
%time hist = model.fit(X_train, y_train, batch_size=2000, epochs = 50, verbose=0)
```

```
CPU times: user 20.4 s, sys: 4.08 s, total: 24.5 s
Wall time: 12 s
```

Model Evaluation

on test set

In [9]:
```
import mlins.evaluation as evaluation

y_test_pred = model.predict(X_test)
metrics_test_single = evaluation.get_metrics_Tarifierung(y_test, y_test_pred.ravel(), verbose=1)
```

Model Evaluation Metrics

Metric	Value
mean squared error	238117600.2100
mean squared relative error	0.3895
max absolute error	211134.1561
max relative error	15.1902
perc. of abs. error above 1	0.9995
perc. of abs. error above 0.1	1.0000
perc. of abs. error above 0.01	1.0000
perc. of rel. error above 0.1	0.9116
perc. of rel. error above 0.05	0.9560
perc. of rel. error above 0.01	0.9913

3. Load and test a better DNN

This model consists of an ensemble of DNNs. It has been improved by several measures and delivers a much better result. Nevertheless is has the same dataset. In addition, training of data is faster.

In [10]:

```
import mlins.lifeFactoryModel as lfm

lfmodel = lfm.LifeFactoryModel.LoadLifeFactoryModel('../Schnittstelle/LFModel_RI_2017_KI.zip')
```

```
INFO:  LOADING LIFE FACTORY MODEL RI_2017_KI SUCCESSFUL
```

In [11]:

```
X_data = df
X_data = X_data.drop(['Beginnmonat','Beginnjahr','GeschlechtVP1'],axis=1)
y_data = X_data['tba']
_, X_test_ens, _, y_test_ens = train_test_split(X_data, y_data, test_size=0.2, random_state=1)
df_leistung_ens = X_test_ens['Leistung']/1e6
X_test_ens = X_test_ens.drop(['Leistung'],axis=1)

y_test_ens_pred = lfmodel.predict('Tba', X_test_ens) * df_leistung_ens

metrics_test_ens = evaluation.get_metrics_Tarifierung(y_test_ens, y_test_ens_pred.ravel(), verbose=0)
from mlins.evaluation import display_metrics_from_dict

my_metrics = {"optimiertes Ensemble": metrics_test_ens,"Einfaches Model": metrics_test_single}
all_metrics = display_metrics_from_dict(metrics=my_metrics)
```

Evaluation Metrics Comparison

Metric	optimiertes Ensemble	Einfaches Model
mean squared error	**67.509896**	238117600.210009
mean squared relative error	**0.000000**	0.389541
max absolute error	**437.962959**	211134.156094
max relative error	**0.002925**	15.190235
perc. of abs. error above 1	**0.267950**	0.999500
perc. of abs. error above 0.1	**0.399950**	1.000000
perc. of abs. error above 0.01	**0.534500**	1.000000
perc. of rel. error above 0.1	**0.000000**	0.911650
perc. of rel. error above 0.05	**0.000000**	0.956000
perc. of rel. error above 0.01	**0.000000**	0.991300

Conclusions and Outlooks

Next steps

- Improve active learning, AutoML and the use of pipelines
- Continue the discussion with the financial market's supervisory authorities

Part II: Volker Dietz will give you deeper insights into the models

Thank you for your attention.

CONTACT

Axel Helmert

Axel.helmert@msg-life.com