Nachvollziehbare Schritte
Pima Indians Diabetes Database

Sanja Srdanovic, 21.01.2022

1. Kurze Darstellung des Problembereichs / Aufriss des Themas

1.1 Inhaltlich

Kern der Untersuchung: Creating a Neural Network with *TensorFlow Keras* for the *Pima Indians Diabetes Database* from *Kaggel*
Grobziele der Arbeit: Inspect the dataset and variables that might have an influence on having diabetes; create an NN model to train the data points to predict whether a person has diabetes or not and evaluate the model

1.2 Begründung des Themas

Darstellung der Relevanz des Themas?

Nowadays, diabetes has been widespread around the globe, so inspecting the variables that influence the occurrence of diabetes with neural networks by predicting if a person has it or not would be of great importance for people to think about their health and to decrease the risk factors as much as it is in their power to avoid getting diabetes.

Darstellung eines persönlichen Erkenntnisinteresses.

This dataset was particularly interesting to me because I have been reading a lot about healthy food and lifestyle. The general picture people have is that only eating sweets causes Insulin resistance or eventually Diabetes, but some people do not consider the fact that there are additional important factors, or better say, the combination of different factors that could lead to such a disease. Therefore, I find it remarkable that people can benefit from research that uses neural networks for detecting and predicting a disease such as diabetes, which is very common nowadays given the hectic lifestyle. With these types of research, we can become more aware of what feature or combination of features leads to diabetes or other diseases.

2. Nachvollziehbare Schritte

2.1 Der Stand der Forschung / Auswertung der vorhandenen Literatur / Tutorials ...

Wurde das Problem früher bereits untersucht?
Welche Aspekte wurden untersucht und welche nicht?
Welche Kontroversen gab es und welche Methoden standen bis jetzt im Vordergrund?
This dataset has been often investigated using different ML models. At this stage of learning, I will use *TensorFlow*, i.e., *Keras* for creating a simple neural network for solving a binary classification problem.

Lösungswege strukturieren!

- Loading and inspecting the data, cleaning the data
- Splitting the data into train and test
- Imputing mean values for training data, dropping NaNs for test data
- Creating a model and training the data
- Plotting the losses x epochs
- Final evaluation of the model

2.2 Fragestellung

1. Can a simple neural network predict if a person has diabetes or not, and with what accuracy?

2.3 Methode

Loading the packages and libraries

I loaded the following packages and libraries: *pandas*, *NumPy*, *matplotlib*, *seaborn*, *Scikit-Learn*, *TensorFlow* and *Keras*.

```
# Import packages and libraries

# pandas, matplotlib, seaborn, numpy
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
# Scikit-Learn:
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn import metrics
# Keras and Tensorflow
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dropout, Dense
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.optimizers import Adam
```

Then, I checked for versions of *TensorFlow* and *Keras*. Respectively, versions 2.4.1 and 2.4.0 are used for the analysis.

```
# Check a version of tensorflow and keras
print(tf.__version__)
print(keras.__version__)
```

Next, I set *TensorFlow* and *NumPy random seed* to get stable, reproducible results.

```
# Set tensorflow and numpy random seed for reproducible results
tf.random.set_seed(42)
np.random.seed(42)
```

Loading and inspecting the dataset

```
# Loading Data:
data = pd.read_csv('diabetes.csv')

# seting printing values to see all data in the console
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)

# data info
print(data.head())
print(data.info())
# 768 entries, 9 columns
# 768 entries and 8 input features (variables) and 1 output feature
print(data.dtypes)
```

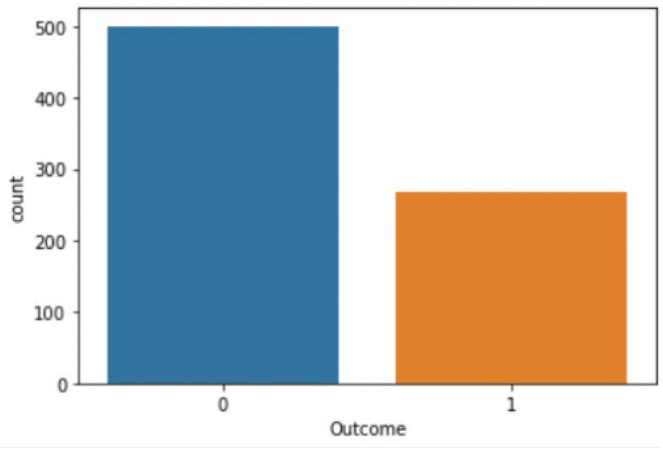
I loaded the dataset '*diabetes'* from CSV. By looking at the *data.head()* or *data.info()*, it can be observed that there are 768 entries and 9 columns. There are 8 numerical variables: *Pregnancies*, *Glucose*, *BloodPressure*, *SkinThickness*, *Insulin*, *BMI*, *DiabetesPedigreeFunction*, and *Age*. The *Outcome* is the categorical value, and it is already encoded: 0 - no diabetes, 1- diabetes.

First visualisations

First, I wanted to have a look at visualisations to get some ideas about the distribution of diabetes and the correlation of variables, so I plotted several figures.

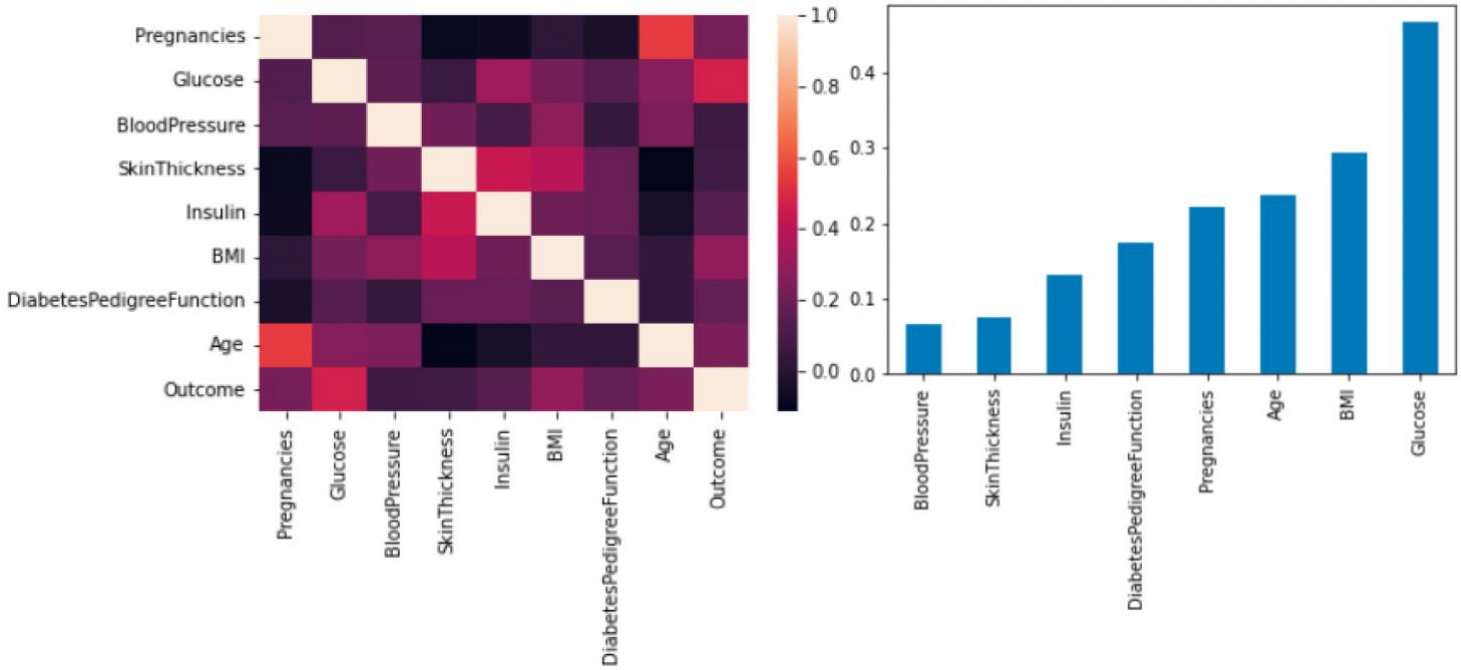
```
"""First visualisations to get some ideas about the distribution of diabetes
   and correlation of variables
"""
# count plot for the Outcome - there is more women with no diabetes than with diabetes
sns.countplot(x='Outcome', data = data)
plt.show()
plt.savefig("Distribution_of_diabetes.png") # save figure
# heatmap to check for the correlation
sns.heatmap(data.corr())
plt.show()
plt.savefig("Correlation_heatmap.png") # save figure
# bar graph to check the correlation by Outcome and
# get some ideas which variables have most influence
data.corr()['Outcome'][:-1].sort_values().plot(kind='bar')
plt.savefig("Correlation_barplot.png") # save figure
# histogram for each variable
data.hist(figsize=(18,12))
plt.show()
plt.savefig("Histograms_variables.png") # save figure
```

By creating a count plot, we can observe that there are more women without diabetes (n=500) than women with diabetes (268).

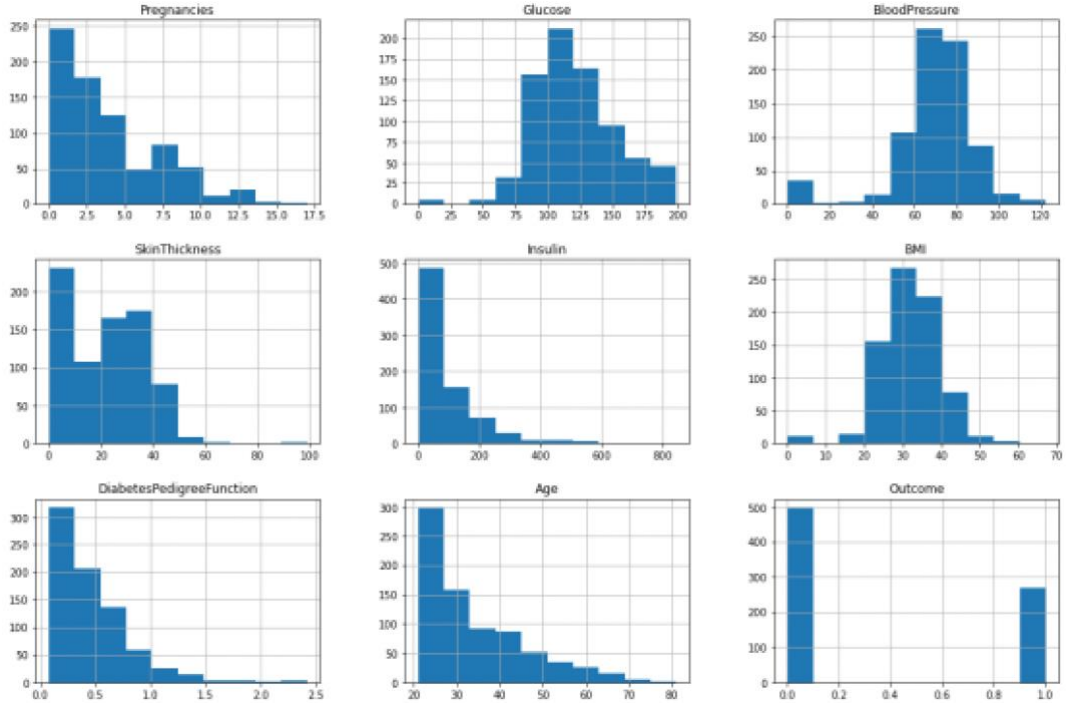


Note. 0 - no diabetes, 1 - diabetes

The heatmap shows the correlation of each variable, and we can see that none of the variables is in a high correlation with the outcome. The highest correlation with the *Outcome* has *Glucose*, which is also illustrated by a bar plot below.



And finally, I plotted a histogram for each variable to observe their distribution.



We can observe that there are also *null values for some features which usually cannot have 0 values.*

Let's inspect the data to check this matter more closely with *data.describe()*, and check if there are some missing values with *data.isnull().sum()* and *data.isna().sum()*.

```
# Data Inspection

print(data.isnull().sum())
print(data.isna().sum())
print(data.describe())
```

```
In [67]: print(data.isna().sum())
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

It seems like there are no zero, i.e., no missing values, but from the distribution plots, we observed that some variables have zero values, which is very unusual. In the next output, we can also notice that there are min values 0 for some of the variables.

```
In [65]: print(data.describe())
       Pregnancies     Glucose  BloodPressure  SkinThickness     Insulin  \
count   768.000000  768.000000     768.000000     768.000000  768.000000
mean      3.845052  120.894531      69.105469      20.536458   79.799479
std       3.369578   31.972618      19.355807      15.952218  115.244002
min       0.000000    0.000000       0.000000       0.000000    0.000000
25%       1.000000   99.000000      62.000000       0.000000    0.000000
50%       3.000000  117.000000      72.000000      23.000000   30.500000
75%       6.000000  140.250000      80.000000      32.000000  127.250000
max      17.000000  199.000000     122.000000      99.000000  846.000000

              BMI  DiabetesPedigreeFunction         Age     Outcome
count  768.000000                768.000000  768.000000  768.000000
mean    31.992578                  0.471876   33.240885    0.348958
std      7.884160                  0.331329   11.760232    0.476951
min      0.000000                  0.078000   21.000000    0.000000
25%     27.300000                  0.243750   24.000000    0.000000
50%     32.000000                  0.372500   29.000000    0.000000
75%     36.600000                  0.626250   41.000000    1.000000
max     67.100000                  2.420000   81.000000    1.000000
```

Particularly, it can be observed that *min* values for *Glucose*, *BloodPressure*, *SkinThickness*, *Insulin*, and *BMI* are 0, which is unlikely to be possible, so this means that these values have not been measured, i.e., these values are missing, so I labelled them as NaN values:

```
conditions_with_nan=['Glucose','BloodPressure','SkinThickness','Insulin','BMI']

for i in conditions_with_nan:
    print("There are " + str(len(data.loc[(data[i]==0),i])) + \
          " instances of value 0 in "+i+" variable.")

for i in conditions_with_nan:
    data.loc[(data[i]==0),i]=np.NaN

print('Missing Number of Observations for all Variables:' + \
      "\n" + str(data.isnull().sum()))

# to check that there are no more 0s as min values
print(data[conditions_with_nan].min())
```

From the following, we can observe how many missing values per variable there are, label them as NaN values and check whether there are still some zero values for the conditions in question by looking at min values of these variables again.

```
In [73]: for i in conditions_with_nan:
    ...:     print("There are " + str(len(data.loc[(data[i]==0),i])) +
variable.")
There are 5 instances of value 0 in Glucose variable.
There are 35 instances of value 0 in BloodPressure variable.
There are 227 instances of value 0 in SkinThickness variable.
There are 374 instances of value 0 in Insulin variable.
There are 11 instances of value 0 in BMI variable.

In [74]: for i in conditions_with_nan:
    ...:     data.loc[(data[i]==0),i]=np.NaN

In [75]: print('Missing Number of Observations for all Variables:' + \
    ...:       "\n" + str(data.isnull().sum()))
Missing Number of Observations for all Variables:
Pregnancies                   0
Glucose                       5
BloodPressure                35
SkinThickness               227
Insulin                     374
BMI                          11
DiabetesPedigreeFunction      0
Age                           0
Outcome                       0
dtype: int64

In [76]: print(data[conditions_with_nan].min())
Glucose          44.0
BloodPressure    24.0
SkinThickness     7.0
Insulin          14.0
BMI              18.2
dtype: float64
```

When I tried dropping the nan values with *data.dropna(inplace=True)* and checked it with *print(data.info()),* only 392 entries were left, which is only 51,04% of the complete dataset. Since the given dataset is already not that big, I thought that just dropping the missing values would leave us with an even smaller dataset, so I decided to come up with a different solution. In this article, I read some of the ways to compensate for missing values in a dataset: https://towardsdatascience.com/6-different-ways-to-compensate-for-missing-values-data-imputation-with-examples-6022d9ca0779. Thus, I decided to go along with the imputation using mean values, but not on the whole dataset. So, I first split the data into training and test, and later I imputed mean values for training data only, and for the test data I did not want to tamper with the data and get an artificial outcome, so I dropped NaN-s and worked with the real data that were left for validation.

First, I divided the data into the input-training (X) and output values-target (y), the input being 8 features that could cause diabetes, and the output is the Outcome (1 - diabetes, 0 - no diabetes).

```
# input values - 8 variables
# output data - outcome
X = data.iloc[:, 0:8].values  # Input values.
y = data.iloc[:, 8].values    # Output values (outcome 1-has diabetes, 0-no diabetes)
```

Then, I split the data into train and test data: 80% for the training data that would be used for the learning phase (614 entries), and the rest 20% for the test data for validation (154 entries).

```
# split data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y,  test_size=0.2, random_state=42)

print('X_train: ' + str(X_train.shape))
print('y_train: ' + str(y_train.shape))
print('X_test: ' + str(X_test.shape))
print('y_test: ' + str(y_test.shape))

print(type(X_test))
```

After I had split the data, I imputed mean values for the missing values for the abovementioned conditions for the training data (X_train) and dropped the NaN values for the test data (X_test).

```
# Impute mean
imp_mean = SimpleImputer(strategy='mean')
imp_mean.fit(X_train)
X_train = imp_mean.transform(X_train)
```

```
# drop the NaN values in X_test
i = pd.isnull(X_test).any(1).nonzero()[0]
y_test_fin=np.delete(y_test, i)

X_test = X_test[~np.isnan(X_test).any(axis=1)]
```

Then I scaled the data, i.e., normalized the input features with *MinMaxScaler* from *Scikit-Learn*. This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g., between zero and one.

```
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Finally, we came to building a model with *Tensorflow Keras*, consisting of an input layer, one hidden layer and an output layer. The activation functions I used for all models was *relu* (Rectified Linear Unit) for the input and the hidden layer, and the *sigmoid* function for the output layer. The optimizer was *Adam*, and for the loss function, I used *BinaryCrossentropy*.

Model1. 64-32-1 no early stopping, no dropout

In an article that investigated this data set, I read that the architecture they used to create a neural network was 64 nodes for the input layer, 34 nodes for the hidden layer and 1 node for the output layer. This is why I decided to start with this architecture, too. And the first model was without early stopping, and without dropout.

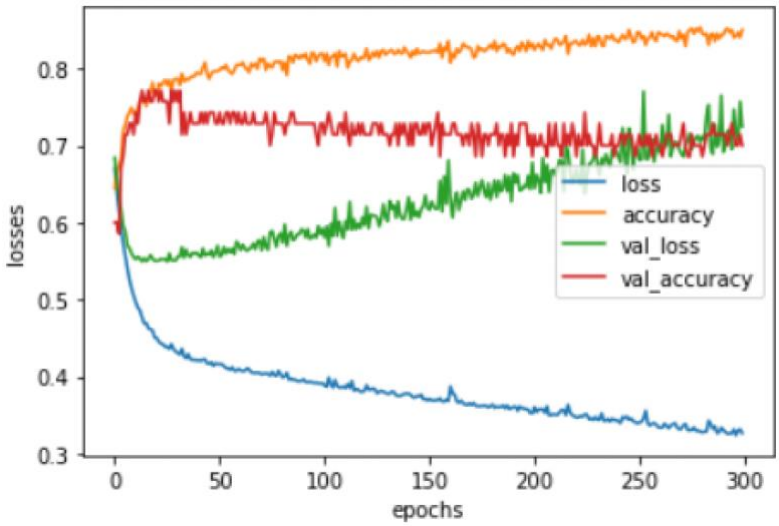
```
model = tf.keras.Sequential([
            tf.keras.layers.Dense(64, input_shape=(8,), activation='relu', kernel_initializer = 'he_normal'),
            tf.keras.layers.Dense(32, activation='relu', kernel_initializer = 'he_normal'),
            tf.keras.layers.Dense(1, activation='sigmoid', kernel_initializer = 'he_normal')
            ])

model.compile(optimizer='Adam',
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=['accuracy'])
```

The model was compiled and trained for 300 epochs. The history object is then converted into a pandas DataFrame and plotted, and we can observe how the *accuracy*, *val_accuracy*, *loss* and *val_loss* change over the 300 epochs.

```
history = model.fit(X_train, y_train, validation_data=(X_test, y_test_fin), epochs=300, verbose = 1)


# Convert history of fitting to pandas Dataframe for plotting
history = pd.DataFrame(history.history)
# Plot losses and accuracy
history.plot(xlabel='epochs', ylabel='losses')
plt.show()
```



From the plot, we can conclude that we are dealing with clear *overfitting*. The accuracy is 70%.

Model 2. 64-32-1 with early stopping, no dropout

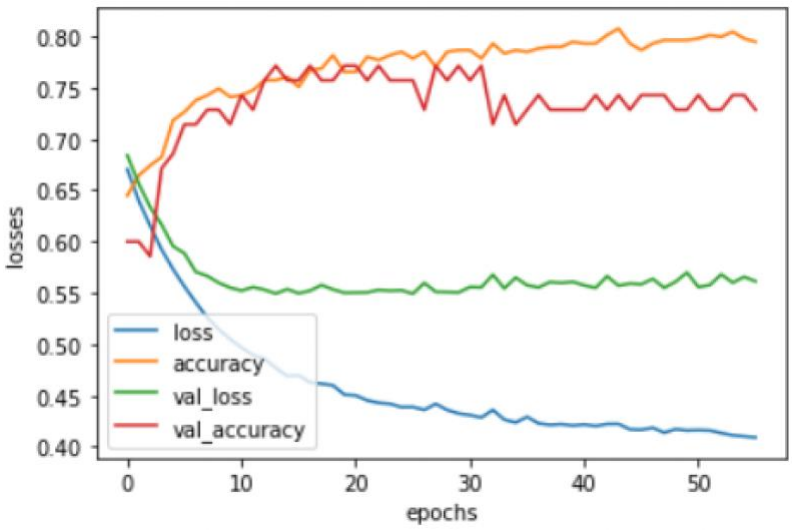
To avoid overfitting, a callback for the training is defined in the next model. The *EarlyStopping* callback from *Keras* is used. Patience is set to 30 epochs and a monitor value is defined as validation loss. This means that the training is aborted if it lasts for 30 epochs there is no improvement in validation loss. The *restore_best_weights* parameter ensures that the model is reset to the state at which the best performance was achieved after training.

```
model = tf.keras.Sequential([
            tf.keras.layers.Dense(64, input_shape=(8,), activation='relu', kernel_initializer = 'he_normal'),
            tf.keras.layers.Dense(32, activation='relu', kernel_initializer = 'he_normal'),
            tf.keras.layers.Dense(1, activation='sigmoid', kernel_initializer = 'he_normal')
            ])

model.compile(optimizer='Adam',
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=['accuracy'])


early_stopping = EarlyStopping(patience=30, monitor='val_loss', restore_best_weights=True)

history = model.fit(X_train, y_train, validation_data=(X_test, y_test_fin), epochs=300, verbose = 1, callbacks=[early_stopping])
```



It can be observed that the training is stopped earlier here because the validation loss gets worse. Also, the accuracy is higher here - 75,7% and the curves look a bit better now.

Model 3. 64-32-1 with early stopping and dropout

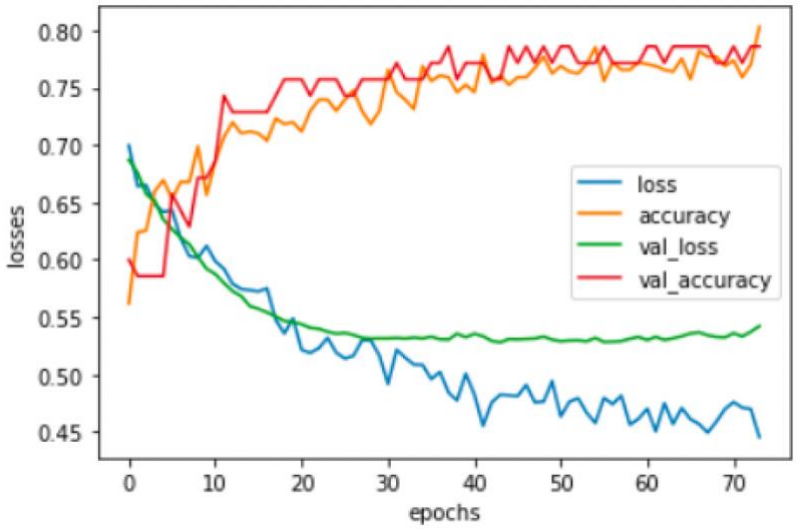
In the next step, dropout is also activated for the model. There will be a 25% dropout rate for the input layer and 50% dropout rate for the hidden layer, which has been shown to be suitable for the increase of the accuracy for the validation set. Again, the *EarlyStopping* callback is used.

```
model = tf.keras.Sequential([
            tf.keras.layers.Dense(64, input_shape=(8,), activation='relu', kernel_initializer = 'he_normal'),
            tf.keras.layers.Dropout(0.25),
            tf.keras.layers.Dense(32, activation='relu', kernel_initializer = 'he_normal'),
            tf.keras.layers.Dropout(0.5),
            tf.keras.layers.Dense(1, activation='sigmoid', kernel_initializer = 'he_normal')
            ])

model.compile(optimizer='Adam',
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=['accuracy'])


early_stopping = EarlyStopping(patience=30, monitor='val_loss', restore_best_weights=True)

history = model.fit(X_train, y_train, validation_data=(X_test, y_test_fin), epochs=300, verbose = 1, callbacks=[early_stopping])
```



Here we get the same accuracy as in the previous model - 75,7%, but the loss looks a bit different.
In the final step, I tried playing around with the number of nodes in the layers to see if the accuracy could be increased even more.

Model 4. Change of number of nodes without dropout 8-32-1

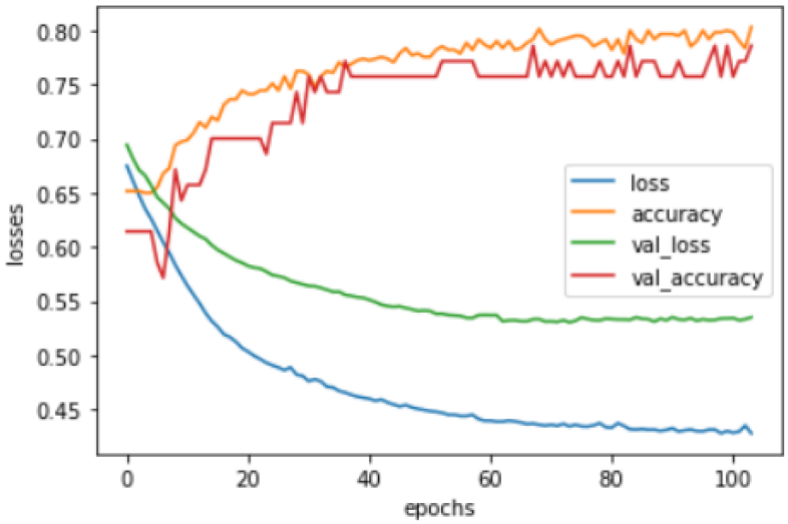
As in the second model, *EarlyStopping* was included, without a dropout and the number of nodes in the input layer was changed to 8.

```
model = tf.keras.Sequential([
            tf.keras.layers.Dense(8, input_shape=(8,), activation='relu', kernel_initializer = 'he_normal'),
            tf.keras.layers.Dense(32, activation='relu', kernel_initializer = 'he_normal'),
            tf.keras.layers.Dense(1, activation='sigmoid', kernel_initializer = 'he_normal')
            ])

model.compile(optimizer='Adam',
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=['accuracy'])


early_stopping = EarlyStopping(patience=30, monitor='val_loss', restore_best_weights=True)

history = model.fit(X_train, y_train, validation_data=(X_test, y_test_fin), epochs=300, verbose = 1, callbacks=[early_stopping])
```



The accuracy is slightly higher, namely 77,1% and the *loss* and *val_loss* look good.

Model5. Change of neuron numbers in layers with dropout 8-32-1

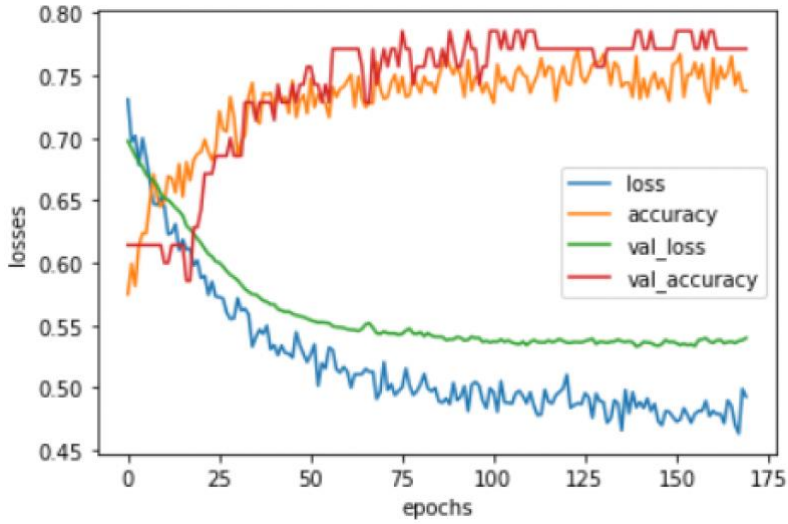
The same number of nodes is used as in the previous model, again the *EarlyStopping* callback was included, but this time the dropouts were included as well.

```
model = tf.keras.Sequential([
            tf.keras.layers.Dense(8, input_shape=(8,), activation='relu', kernel_initializer = 'he_normal'),
            tf.keras.layers.Dropout(0.25),
            tf.keras.layers.Dense(32, activation='relu', kernel_initializer = 'he_normal'),
            tf.keras.layers.Dropout(0.5),
            tf.keras.layers.Dense(1, activation='sigmoid', kernel_initializer = 'he_normal')
            ])

model.compile(optimizer='Adam',
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=['accuracy'])


early_stopping = EarlyStopping(patience=30, monitor='val_loss', restore_best_weights=True)

history = model.fit(X_train, y_train, validation_data=(X_test, y_test_fin), epochs=300, verbose = 1, callbacks=[early_stopping])
```



In this model, the data is trained for a higher number of epochs compared to previous models, but it gets the highest accuracy - 78,6%.

2.6 Ergebnisse

By comparing different models, it can be concluded that we obtained better results when using *EarlyStopping* callback and *Dropout* layers which prevent overfitting. Additionally, the number of neurons in the layers can change the accuracy, because as models with too few neurons are not always good, neither is having too many neurons, so the optimal number should be detected.

The architecture of the models can be output with the summary method - *print(model.summary()).*

```
In [99]: print(model.summary())
Model: "sequential_72"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_215 (Dense)            (None, 8)                 72
_________________________________________________________________
dropout_86 (Dropout)         (None, 8)                 0
_________________________________________________________________
dense_216 (Dense)            (None, 32)                288
_________________________________________________________________
dropout_87 (Dropout)         (None, 32)                0
_________________________________________________________________
dense_217 (Dense)            (None, 1)                 33
=================================================================
Total params: 393
Trainable params: 393
Non-trainable params: 0
_________________________________________________________________
None
```

The final evaluation is calculated on the test set.

```
# Final evaluation of generalisation error:
# Calculate accuracy on test set and print for final evaluation
acc_test = model.evaluate(X_test, y_test_fin)[1]
print(f'Accuracy Test Set      : {acc_test*1E2:.1f}%')
```

In the Table below, I will provide an overview of the accuracy of each of the models presented.

Model	**Model1**	**Model2**	**Model3**	**Model4**	**Model5**
Structure	64-32-1 No early stopping no dropout	64-32-1 Early stopping No dropout	64-32-1 Early stopping dropout	8-32-1 Early stopping No dropout	8-32-1 Early stopping Dropout
Accuracy	70 %	75.7 %	75.7 %	77.1 %	**78.6 %**

So, we can conclude that the last model with *EarlyStopping*, *Dropouts* and the change of the number of neurons to 8 in the first, input layer, 32 in the hidden layer and 1 in the output layer is the best one, as it yields the accuracy of 78.6 %.

With *model.predict()*, we can create *y_pred_cat* to predict whether the outcome of diabetes is 1 (diabetes) or 0 (no diabetes), based on the 8 variables for the data in *X_test*, and then compare the outcome data in *y_test_fin* with predicted data in *y_pred_cat*.

```
# compare y_test_fin and y_pred_cat
y_pred=model.predict(X_test)
y_pred_cat = np.around(y_pred)
```

We can also check the predictions and accuracy of the model with the *Confusion Matrix* from *Scikit-Learn*, where we compare actual data and data predicted by the model.

```
# Confusion Matrix

cm = confusion_matrix(y_test_fin, y_pred_cat)
print(cm)
```

```
Output from spyder call 'get_namespace_view':
[[36  7]
 [ 8 19]]
```

The algorithm correctly predicted 36 patients to be truly diabetic negative (true negative), but it falsely predicted 7 patients to be diabetic positive (false positive). Then, it falsely predicted for 8 people that they are diabetes negative (false negative) and truly predicted 19 people to be diabetic positive (true positive). The test set contained 70 patients in total (because the missing values were removed for the test data). In total, the model predicted 44 negatives, but there were 43 actual total negatives. The model predicted 26 to be positive, but there were 27 actual positives.
For the final model, we got an accuracy of 78,6, which can be also corroborated from the Confusion Matrix as well: (36+19)/70 *100 = 78,6.

	precision	recall	f1-score	support
0	0.82	0.84	0.83	43
1	0.73	0.70	0.72	27
accuracy			0.79	70
macro avg	0.77	0.77	0.77	70
weighted avg	0.78	0.79	0.78	70

From the classification report, it can be read out that the precision of negative diabetes outcomes is 82% and for positive ones is 73%. The recall is 84% for no diabetes, and 70% for diabetes, and the f1-score is 83% and 72% respectively. The overall accuracy is as already mentioned 78,6 or here it is rounded to 79%.

2.7 Ausblick

Using a simple neural network and the tools to avoid overfitting, it was possible to create a model to classify the data into healthy patients and patients with diabetes, with quite a good accuracy of 78.6%. What could be done in further projects to improve the quality and accuracy of the neural network model is to use other more fine-grained Machine Learning methods. Additionally, increasing the data set could provide an even better outcome. Even though the data set with 768 entries is big enough for machine learning, there are a lot of missing values for the important variables that can cause diabetes. Providing more data points with all measures without missing values would be more suitable for the data training and would provide more accurate results and predictions.
The procedure described here, however, is a good starting point for practising how to create neural networks that could help us detect the risk factors for diabetes and predict if a person could get it based on their health factors, and a comparable model could be applied to other datasets with a similar topic.