Schwingungberechnung mit Neuralnetzwerk

Die Aufgabe ist die automatisierte Ermittlung der Stärke von Schwingungen in Krananlagen mithilfe eines Neuralnetzwerks.

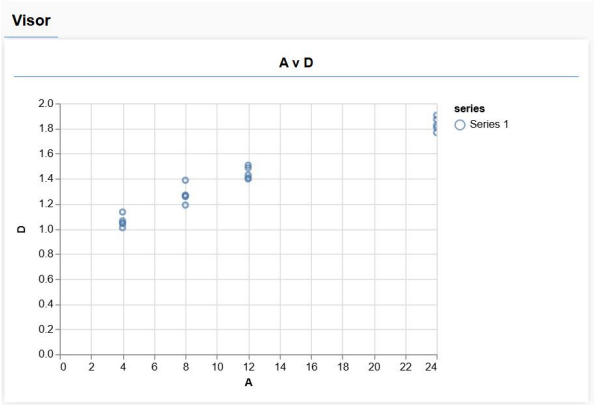
Ein Neuralnetzwerk wird meistens eingesetzt, wo viele Werte gleichzeitig und schnell bewertet werden müssen. Es gibt 2 hauptsächliche Aufgaben von einem Neuralnetzwerk - Klassifizierung und Regression. Bei einer Klassifizierung wird versucht zu schätzen, was die wahrscheinlichkeit ist, dass anhand von Eingabewerten, etwas wahr ist oder zu einer Gruppe gehört z.B Bild von Hund oder Katze. Bei einer Regression wird versucht zu schätzen, was der reele Zahlenwert von etwas ist z.B Preis von einem Haus nach 20 Jahren, Temperatur.

Zur Lösung der Aufgabe wurden die Daten in der folgenden Tabelle als Beispieldaten erhalten. A,b,c sind Eingaben und d ist die Ausgabe.

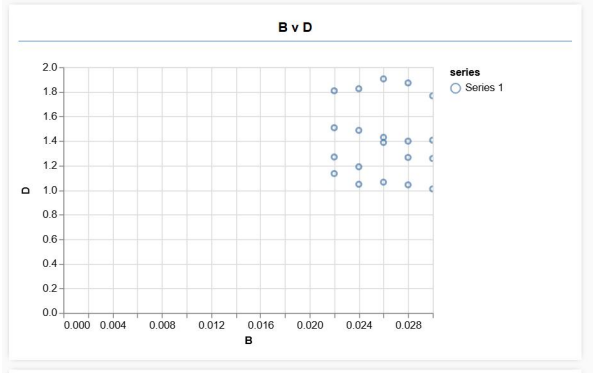
a	b	c	d
4	0.03	0	1.134
4	0.028	0	1.104
4	0.026	0	1.171
4	0.024	0	1.242
4	0.022	0	1.174
8	0.03	0	1.233
8	0.028	0	1.323
8	0.026	0	1.372
8	0.024	0	1.248
8	0.022	0	1.199
12	0.03	0	1.502
12	0.028	0	1.328
12	0.026	0	1.479
12	0.024	0	1.386
12	0.022	0	1.302
24	0.03	0	1.719
24	0.028	0	1.785
24	0.026	0	1.942
24	0.024	0	1.957
24	0.022	0	1.963
4	0.03	1	1.009
4	0.028	1	1.042
4	0.026	1	1.064
4	0.024	1	1.047
4	0.022	1	1.134
8	0.03	1	1.257
8	0.028	1	1.265
8	0.026	1	1.387
8	0.024	1	1.189
8	0.022	1	1.269
12	0.03	1	1.406
12	0.028	1	1.398
12	0.026	1	1.429
12	0.024	1	1.486
12	0.022	1	1.507

24	0.03	1	1.767
24	0.028	1	1.872
24	0.026	1	1.905
24	0.024	1	1.825
24	0.022	1	1.808

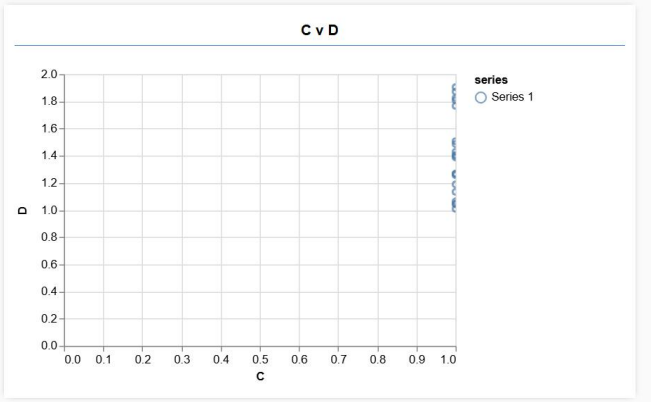
Graphen



Graph von a vs d



Graph von b vs d



Graph von c vs d

Vorgang

Zur Lösung der Aufgabe wurde so angegangen:

Schritt 1: Die Daten werden zuerst geshuffelt - random neugeordnet - und danach in 3 Gruppen geteilt.
Die neue Zuordnung dient dazu, dass unser Netzwerk nicht immer die selben Daten bekommt bzw sich nicht an einer spezifischen Reihenfolge gewohnt.

1. Trainingsdaten: Ein Teil der gelieferten Daten wird zum Trainen unseres Neuralnetzwerks verwendet. Damit soll unser Netzwerk lernen und eine mathematische Beziehung zwischen den Daten finden.
2. Validierungsdaten: Ein Teil der Daten wird zur Validierung verwendet. Wir liefern unserem Netzwerk einige Eingabe und vergleichen die Ergebnisse aus dem Netwerk mit den Ergebnisse aus der Validierungsdaten (die 'd' Spalten).
3. Testdaten: in den meisten Fälle lässt die Testdaten von dem Rechner zufällig generieren. Für unserem Fall werden wir jedoch ein Teil der Daten verwenden.

Schritt 2: Danach werden die Daten vorbearbeitet. Das bedeutet, dass wir versuchen, unsere Werte so zu gestalten, dass sie zwischen kleineren Werte liegen z. B wenn unsere Werte von einer Spalte zwischen -24000 und 24000 steuern würden, müssen wir das auf einen Bereich von -1 bis 1 reduzieren. So ist es einfacher und schneller für das Netzwerk zu rechnen. Dieser Prozess heißt Normalisierung.
Dafür gibt es verschiedenen Methoden. Eine davon ist alle Werte durch den maximalen Wert zu dividieren. In unserem Beispielfall würden die Werte dann statt zwischen -24000 und 24000 zu liegen, dann nur zwischen -1 und 1 liegen - angenommen, dass 24000 der maximale Wert wäre und wir dividieren dadurch. Ein anderer Weg ist mithilfe der Standardabweichung (Standarddeviation auf English).

Programmiersprachen:

Im Projekt werden 2 Programmiersprachen eingesetzt.

1. **Python**: wobei verschiedene öffentlich verfügbare Bibliotheken eingesetzt und Algorithmen entwurfen wurden wie SciKit-learning, Tensorflow, Matplotlib, numpy, pandas und Keras. Das Ziel des Einsatzes von verschiedenen Lösungswegen ist herauszufinden, mit welchem Weg oder Algorithmus man die niedrigste Abweichung von erwarteten Werte bekommt.
2. **Javascript**: Mithilfe von TensorflowJS.

Python:
Für das Projekt wird ein Jupyterbook verwendet. Dadurch können wir das Programm bei der Entwicklung in Abschnitte teilen und diese testen und Ergebnisse sehen. So müssen wir nicht immer das ganze Program neustarten.

```
import pandas as pd

structure_data = pd.read_csv('./train.csv')
structure_data
```

✓ 0.0s

	a	b	c	d
0	4	0.030	0	1.134
1	4	0.028	0	1.104
2	4	0.026	0	1.171
3	4	0.024	0	1.242
4	4	0.022	0	1.174
5	8	0.030	0	1.233
6	8	0.028	0	1.323
7	8	0.026	0	1.372
8	8	0.024	0	1.248
9	8	0.022	0	1.199
10	12	0.030	0	1.502
11	12	0.028	0	1.328
12	12	0.026	0	1.479
13	12	0.024	0	1.386

Auslesen der Daten aus der CSV Datei mit Panda

```
structure_data_shuffled = structure_data.sample(n=len(structure_data), random_state=1)
structure_data_shuffled
```

✓ 0.0s

	a	b	c	d
2	4	0.026	0	1.171
31	12	0.028	1	1.398
3	4	0.024	0	1.242
21	4	0.028	1	1.042
27	8	0.026	1	1.387
29	8	0.022	1	1.269
22	4	0.026	1	1.064
39	24	0.022	1	1.808
19	24	0.022	0	1.963
26	8	0.028	1	1.265
32	12	0.026	1	1.429
17	24	0.026	0	1.942
30	12	0.030	1	1.406

Neuzuordnung der Daten. Zum Erkennen ist das die 31. Reihe jetzt die 2. Reihe geworden ist.

Wir trennen jetzt die Daten in Test-, Validierungs- und Trainingsdaten. Die ersten 20 Reihen unseres neuen Datensatz sind Trainingsdaten, die nächsten 10 sind Testdaten und die restlichen 10 sind Validierungsdaten.

```
#split the data into test, trin and validation data
train_pd, test_pd, val_pd = structure_data_shuffled[:20], structure_data_shuffled[20:30], structure_data_shuffled[30:]
len(train_pd), len(test_pd), len(val_pd)
```

✓ 0.0s Python

```
(20, 10, 10)
```

```
# try to get the data from the separeted data
X_train, y_train = train_pd.to_numpy()[:, :-1], train_pd.to_numpy()[:, -1]
X_val, y_val = val_pd.to_numpy()[:, :-1], val_pd.to_numpy()[:, -1]
X_test, y_test = test_pd.to_numpy()[:, :-1], test_pd.to_numpy()[:, -1]

X_train.shape, y_train.shape, X_val.shape, y_val.shape, X_test.shape, y_test.shape
```

✓ 0.0s Python

```
((20, 3), (20,), (10, 3), (10,), (10, 3), (10,))
```

Trennen von Daten in Gruppen

In der 2. Sektion trennen wir wieder die Daten in Eingaben (X_* = a-c Spalten) und Ausgaben (y_* = d-Spalte). Unser X_train enthält dann z.B die Spalten a, b, c der ersten 20 Reihen und y_train die Spalte d der ersten 20 Reihen.

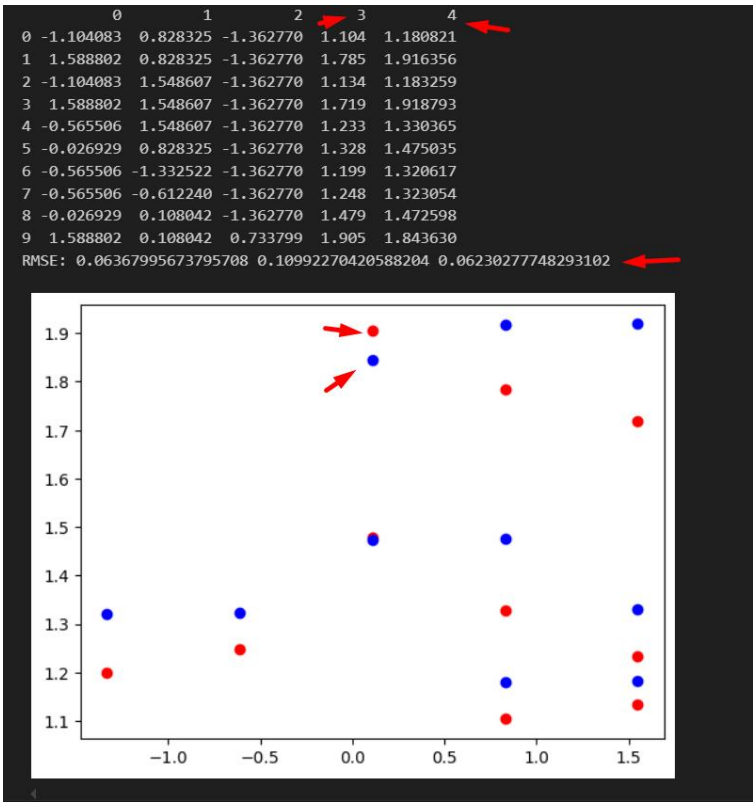
```python
# preprocessing with standard deviation to normalise data
from sklearn.preprocessing import StandardScaler
import numpy as np
import matplotlib


scaler = StandardScaler().fit(X_train[:, :3])

def preprocessor(X):
  A = np.copy(X)
  A[:, :3] = scaler.transform(A[:, :3])
  return A

X_train, X_val, X_test = preprocessor(X_train), preprocessor(X_val), preprocessor(X_test)
```

✓ 0.0s

Normalisierung der Daten mithilfe der Standardscaler Funktion aus der Sci-Kit Bibliothek

Ergebnisse ablesen:

Um die Ablesung zu vereinfachen, werden die Daten wie folgt immer dargestellt:



	0	1	2	3	4
0	-1.104083	0.828325	-1.362770	1.104	1.180821
1	1.588802	0.828325	-1.362770	1.785	1.916356
2	-1.104083	1.548607	-1.362770	1.134	1.183259
3	1.588802	1.548607	-1.362770	1.719	1.918793
4	-0.565506	1.548607	-1.362770	1.233	1.330365
5	-0.026929	0.828325	-1.362770	1.328	1.475035
6	-0.565506	-1.332522	-1.362770	1.199	1.320617
7	-0.565506	-0.612240	-1.362770	1.248	1.323054
8	-0.026929	0.108042	-1.362770	1.479	1.472598
9	1.588802	0.108042	0.733799	1.905	1.843630

RMSE: 0.06367995673795708 0.10992270420588204 0.06230277748293102

Daten ablesen

Spalten 0 - 2 der Tabelle sind die normalisierten Eingaben. A-c
Spalte 3 sind die zu erwartenden Ergebnisse. d
Spalte 4 sind die vom Netzwerk erratenen Werte von dem Netzwerk, d_pred

Weiterhin sind die RMSE (Root mean squared Error) aufgelistet von den erratenen Werten. Dieser Wert ist die Abweichung von d_pred von d und bezeichnet wie gut unser Netzwerk die Werte erraten konnte. Bei einer RMSE von 0 hat unser Netzwerk keinen Fehler gemacht.
Der erste RMSE Wert ist der der Trainingsdaten, gefolgt von dem der Validierungsdaten und abschließend der der Testdaten.

Wichtig ist, dass diese Werte niedrig sind aber nicht zu weit von einandern liegen. Wenn der Wert zu weit von einander liegt, spricht man von overfitting (Überspezialisierung). Also dann kann unser Netzwerk einige Werte besser erraten als die anderen. Unser Ziel ist alles so gut wie möglich zu erraten. Z.B wenn der RMSE Wert von den Trainingsdaten 0.0003 ist aber der RMSE Wert von den Validierungsdaten 3.000 ist, hat unser Netzwerk sich zu sehr auf die Trainingsdaten spezialisiert, da er bei neuen Daten zu viele Fehler macht. Dieser Unterschied wird in diesem Bericht RMSE Unterschied bezeichnet.

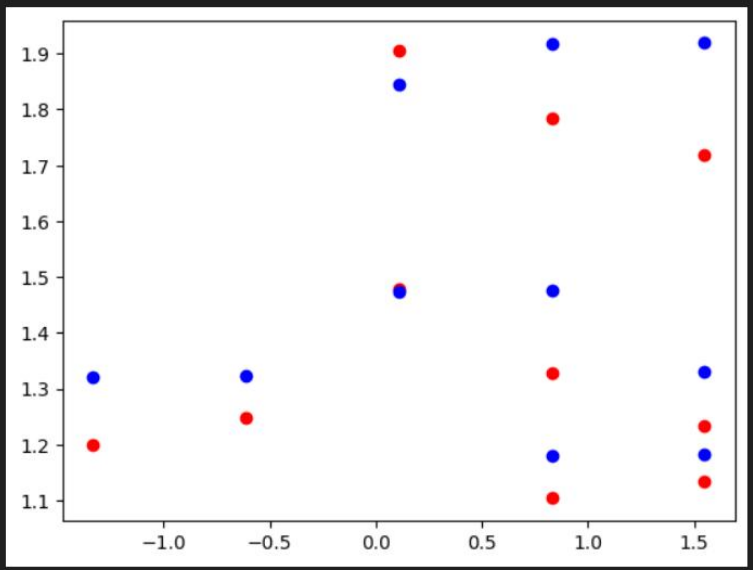
Zuletzt ist dann ein Scatterplot von den zu erwartenden Ergebnisse (rot) und die vom Netzwerk erratenen Werte (blau) zu sehen. So kann man sehen wie weit die Werte von einandern abweichen. Hier wird immer der b-wert gegen y und y_pred der Validierungsdaten visualisiert.

Algorithmen:

Linear regression:
Unser erster Algorithmus ist die Linearregression. Die RMSE sind in Ordnung und die ausgelieferte Ergebnisse sind fast gleich. Der RMSE Unterschied der y_train (0.06) und y_val (0.1) ist jedoch zu hoch - liegt aktuell zwischen 0.01 und 0.04 von einander. Unser Netzwerk müsste sich über-spezialisiert haben aber weil die Wert aller RMSE sehr niedrig sind, ist das ein gutes Netzwerk.

```
# try to train with different training models
from sklearn.metrics import mean_squared_error as mse
from sklearn.metrics import root_mean_squared_error as rmse
from sklearn.linear_model import LinearRegression

lm = LinearRegression().fit(X_train, y_train)
y_train_pred=lm.predict(X_train)
y_val_pred=lm.predict(X_val)
y_test_pred=lm.predict(X_test)

#visualisation
X_visual=X_val
Y_visual=y_val
Y_visual_pred =y_val_pred
pyplot.scatter(X_visual[:, 1], Y_visual, c="red")
pyplot.scatter(X_visual[:, 1], Y_visual_pred, c="blue")
lsit = merge_data(X_visual,Y_visual, Y_visual_pred)
print(pd.DataFrame(lsit))
rmse(y_train_pred, y_train), rmse(y_val_pred, y_val), rmse(y_test_pred, y_test)

✓ 0.3s
```

```
          0         1         2     3         4
0 -1.104083  0.828325 -1.362770  1.104  1.180821
1  1.588802  0.828325 -1.362770  1.785  1.916356
2 -1.104083  1.548607 -1.362770  1.134  1.183259
3  1.588802  1.548607 -1.362770  1.719  1.918793
4 -0.565506  1.548607 -1.362770  1.233  1.330365
5 -0.026929  0.828325 -1.362770  1.328  1.475035
6 -0.565506 -1.332522 -1.362770  1.199  1.320617
7 -0.565506 -0.612240 -1.362770  1.248  1.323054
8 -0.026929  0.108042 -1.362770  1.479  1.472598
9  1.588802  0.108042  0.733799  1.905  1.843630
RMSE: 0.06367995673795708 0.10992270420588204 0.06230277748293102
```



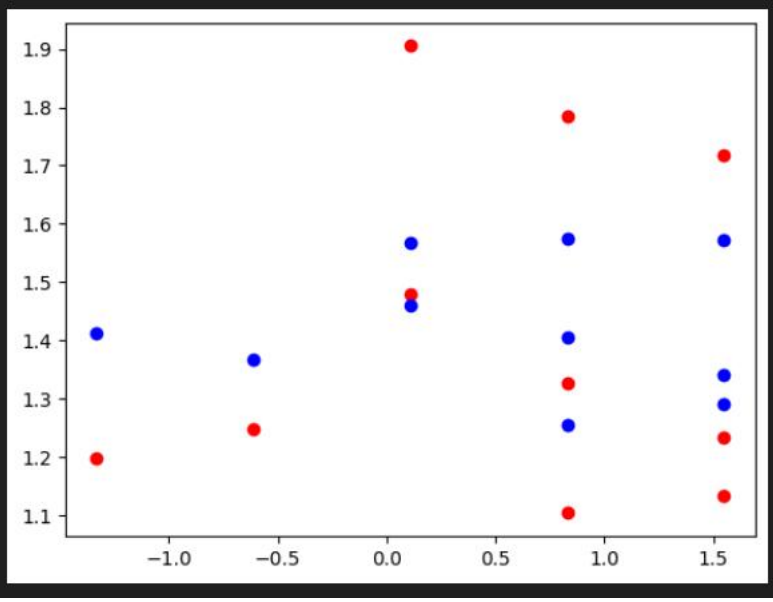
KNeighbor Regression:
Bei der KNeighbors Regression werden die Werte anhand benachbarter Werte geschätzt. Durch das Anpassen des variabels n_neighbors kann man einen niedrigeren oder höheren RMSE Wert erzielen. Jedoch besteht die Gefahr eines Overfitting. Bei unseren Daten war 10 der beste Wert.

Die RMSE Werte sind hier zwar höher als bei der Linearregression (0.17 bis 0.19). Die RMSE Unterschied ist jedoch nicht zu hoch - zwischen 0.01 und 0.02 von einandern.

```python
from sklearn.neighbors import KNeighborsRegressor

knn = KNeighborsRegressor(n_neighbors=10).fit(X_train, y_train)
#prediction
y_train_pred=knn.predict(X_train)
y_val_pred=knn.predict(X_val)
y_test_pred=knn.predict(X_test)

#visualisation
X_visual=X_val
Y_visual=y_val
Y_visual_pred =y_val_pred
pyplot.scatter(X_visual[:, 1], Y_visual, c="red")
pyplot.scatter(X_visual[:, 1], Y_visual_pred, c="blue")
lsit = merge_data(X_visual,Y_visual, Y_visual_pred)
print(pd.DataFrame(lsit))
print("\nRMSE:", 'train:', rmse(y_train_pred, y_train), ', valid:', rmse(y_val_pred, y_val),', test:', rmse(y_test_pred, y_test))
```

✓ 0.5s

```
          0         1         2      3       4
0 -1.104083  0.828325 -1.362770  1.104  1.2547
1  1.588802  0.828325 -1.362770  1.785  1.5752
2 -1.104083  1.548607 -1.362770  1.134  1.2906
3  1.588802  1.548607 -1.362770  1.719  1.5715
4 -0.565506  1.548607 -1.362770  1.233  1.3419
5 -0.026929  0.828325 -1.362770  1.328  1.4044
6 -0.565506 -1.332522 -1.362770  1.199  1.4125
7 -0.565506 -0.612240 -1.362770  1.248  1.3664
8 -0.026929  0.108042 -1.362770  1.479  1.4598
9  1.588802  0.108042  0.733799  1.905  1.5684

RMSE: train: 0.18824054690740782 , valid: 0.17434779035020778 , test: 0.19616407163392596
```

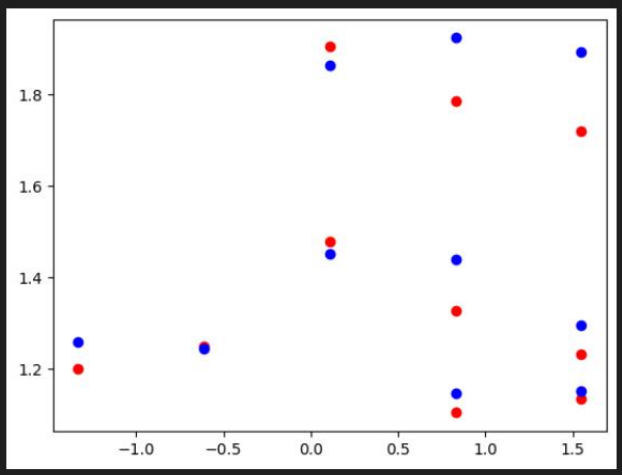


Randomforest Regression:
RMSE Wert zwischen 0.03 bis 0.08. RMSE Unterschied zwischen 0.03 und 0.05.

```python
from sklearn.ensemble import RandomForestRegressor

rfr = RandomForestRegressor(max_depth=10, n_estimators=1000).fit(X_train, y_train)
#prediction
y_train_pred=rfr.predict(X_train)
y_val_pred=rfr.predict(X_val)
y_test_pred=rfr.predict(X_test)

#visualisation
X_visual=X_val
Y_visual=y_val
Y_visual_pred =y_val_pred
pyplot.scatter(X_visual[:, 1], Y_visual, c="red")
pyplot.scatter(X_visual[:, 1], Y_visual_pred, c="blue")
lsit = merge_data(X_visual,Y_visual, Y_visual_pred)
print(pd.DataFrame(lsit))
print("\nRMSE:", 'train:', rmse(y_train_pred, y_train), ', valid:', rmse(y_val_pred, y_val),', test:', rmse(y_test_pred, y_test))
```

✓ 2.5s

```
          0         1         2     3         4
0 -1.104083  0.828325 -1.362770  1.104  1.147418
1  1.588802  0.828325 -1.362770  1.785  1.922857
2 -1.104083  1.548607 -1.362770  1.134  1.150867
3  1.588802  1.548607 -1.362770  1.719  1.893313
4 -0.565506  1.548607 -1.362770  1.233  1.295127
5 -0.026929  0.828325 -1.362770  1.328  1.439503
6 -0.565506 -1.332522 -1.362770  1.199  1.257487
7 -0.565506 -0.612240 -1.362770  1.248  1.243748
8 -0.026929  0.108042 -1.362770  1.479  1.452242
9  1.588802  0.108042  0.733799  1.905  1.863540

RMSE: train: 0.03506988910590026 , valid: 0.08586361674539518 , test: 0.06252428376159104
```



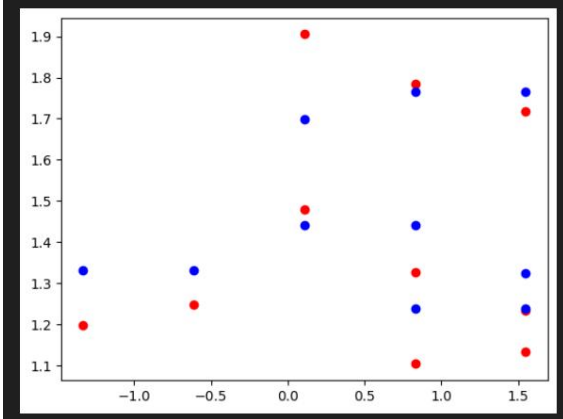
Gradientboosting Regression:

RMSE zwischen 0.1 und 0.13. RMSE Unterschied zwischen 0.01 und 0.02.

```python
from sklearn.ensemble import GradientBoostingRegressor

gbr = GradientBoostingRegressor(n_estimators=10, learning_rate=0.1).fit(X_train, y_train)
#prediction
y_train_pred=gbr.predict(X_train)
y_val_pred=gbr.predict(X_val)
y_test_pred=gbr.predict(X_test)

#visualisation
X_visual=X_val
Y_visual=y_val
Y_visual_pred =y_val_pred
pyplot.scatter(X_visual[:, 1], Y_visual, c="red")
pyplot.scatter(X_visual[:, 1], Y_visual_pred, c="blue")
lsit = merge_data(X_visual,Y_visual, Y_visual_pred)
print(pd.DataFrame(lsit))
print("\nRMSE:", 'train:', rmse(y_train_pred, y_train), ', valid:', rmse(y_val_pred, y_val),', test:', rmse(y_test_pred, y_test))
```

✓ 0.5s

```
          0         1         2     3         4
0 -1.104083  0.828325 -1.362770  1.104  1.238800
1  1.588802  0.828325 -1.362770  1.785  1.764836
2 -1.104083  1.548607 -1.362770  1.134  1.238800
3  1.588802  1.548607 -1.362770  1.719  1.764836
4 -0.565506  1.548607 -1.362770  1.233  1.323888
5 -0.026929  0.828325 -1.362770  1.328  1.440609
6 -0.565506 -1.332522 -1.362770  1.199  1.330517
7 -0.565506 -0.612240 -1.362770  1.248  1.330517
8 -0.026929  0.108042 -1.362770  1.479  1.440609
9  1.588802  0.108042  0.733799  1.905  1.698589

RMSE: train: 0.11065225658366981 , valid: 0.10990367761586725 , test: 0.13701253749645487
```



Tensorflow
Weitere Algorithmen werden mithilfe von der Tensorflow Bibliothek implementiert. Im Gegenteil zur Sci-Kit Learn Bibliothek können wir das je nach Wünsch strenger oder komplizierter machen. Je komplizierter desto besser, aber es besteht eine höhere Gefahr eines Overfittings.

Einfache Regression:
Hier hat unser Netzwerk die Möglichkeit 300 mal durch die Datensätze zu gehen und sich zu verbessern. Dieses Zyklus nennt man ein **Epoch**. Hier verwenden wir eine ReLU (Rectified Linear Unit) und ein Linearregression Funktion. Eine ReLU Funktion nimmt einen Wert an und gibt den Wert 0 aus, wenn der Wert negativ ist und der Wert zurück wenn nicht negativ. Danach speichern wir das in eine .keras Datei.

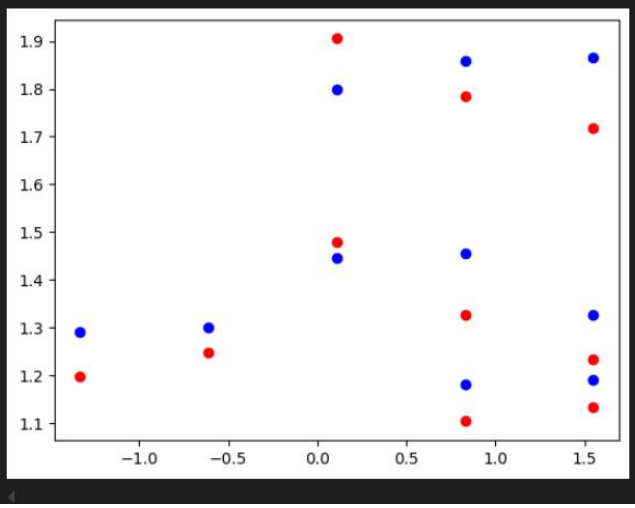
Der RMSE Wert ist hier der kleinste und hat auch der niedrigste RMSE Unterschied. RMSE Wert von 0.07 bis 0.09 und RMSE Unterschied von 0.01 bis 0.02.

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import *
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.metrics import RootMeanSquaredError
from tensorflow.keras.optimizers import Adam

simple_nn = Sequential()
simple_nn.add(InputLayer((3,)))
simple_nn.add(Dense(2, 'relu'))
simple_nn.add(Dense(1, 'linear'))

opt = Adam(learning_rate=.1)
cp = ModelCheckpoint('models/simple_nn.keras', save_best_only=True)
simple_nn.compile(optimizer=opt, loss='mse', metrics=[RootMeanSquaredError()])
simple_nn.fit(x=X_train, y=y_train, validation_data=(X_val, y_val), callbacks=[cp], epochs=300)
```

```python
from tensorflow.keras.models import load_model

simple_nn = load_model('models/simple_nn.keras')
#prediction
y_train_pred=simple_nn.predict(X_train)
y_val_pred=simple_nn.predict(X_val)
y_test_pred=simple_nn.predict(X_test)

#visualisation
X_visual=X_val
Y_visual=y_val
Y_visual_pred =y_val_pred
pyplot.scatter(X_visual[:, 1], Y_visual, c="red")
pyplot.scatter(X_visual[:, 1], Y_visual_pred, c="blue")
lsit = merge_data(X_visual,Y_visual, Y_visual_pred)
print(pd.DataFrame(lsit))
print("\nRMSE:", 'train:', rmse(y_train_pred, y_train), ', valid:', rmse(y_val_pred, y_val),', test:', rmse(y_test_pred, y_test))
```

✓ 1.1s

```
          0         1         2     3           4
0 -1.104083  0.828325 -1.362770  1.104  [1.1815715]
1  1.588802  0.828325 -1.362770  1.785  [1.8580818]
2 -1.104083  1.548607 -1.362770  1.134  [1.1902896]
3  1.588802  1.548607 -1.362770  1.719   [1.864506]
4 -0.565506  1.548607 -1.362770  1.233   [1.326948]
5 -0.026929  0.828325 -1.362770  1.328  [1.4548883]
6 -0.565506 -1.332522 -1.362770  1.199  [1.2920754]
7 -0.565506 -0.612240 -1.362770  1.248  [1.3007936]
8 -0.026929  0.108042 -1.362770  1.479  [1.4461703]
9  1.588802  0.108042  0.733799  1.905  [1.7993965]

RMSE: train: 0.07043611253060633 , valid: 0.09181817630478213 , test: 0.07684602303512071
```



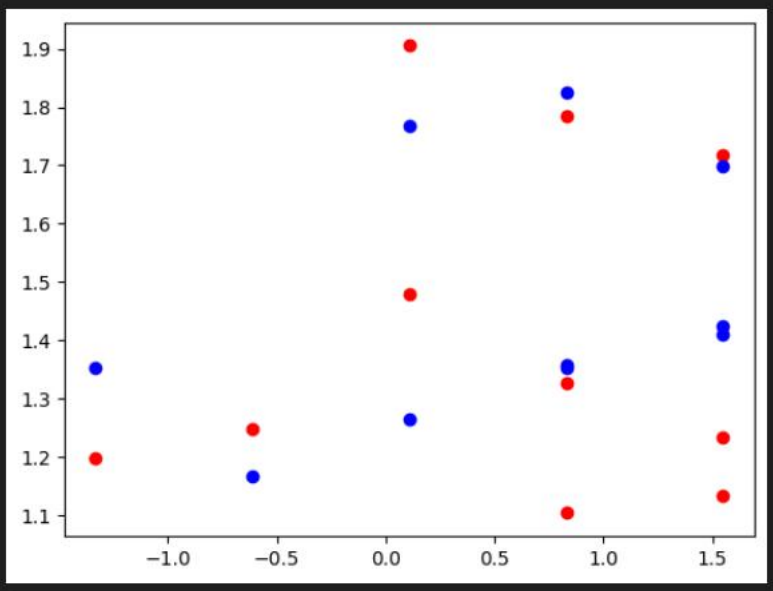
Kompliziertere Regression:
Hier ist genau das Gleiche wie die einfache Regression, jedoch setzen wir 4 mal mehr ReLU Funktionen ein plus ein Epoch von 100. Wir bekommen zwar einen niedrigen RMSE Wert fur die Trainingsdaten. Der RMSE Unterschied ist jedoch zu gross - 0.10. Hier kann man von Overfitting reden.

```python
large_nn = Sequential()
large_nn.add(InputLayer((3,)))
large_nn.add(Dense(256, 'relu'))
large_nn.add(Dense(128, 'relu'))
large_nn.add(Dense(64, 'relu'))
large_nn.add(Dense(32, 'relu'))
large_nn.add(Dense(1, 'linear'))

opt = Adam(learning_rate=.1)
cp = ModelCheckpoint('models/large_nn.keras', save_best_only=True)
large_nn.compile(optimizer=opt, loss='mse', metrics=[RootMeanSquaredError()])
large_nn.fit(x=X_train, y=y_train, validation_data=(X_val, y_val), callbacks=[cp], epochs=100)
```

✓ 23.0s

```
large_nn = load_model('models/large_nn.keras')
#prediction
y_train_pred=large_nn.predict(X_train)
y_val_pred=large_nn.predict(X_val)
y_test_pred=large_nn.predict(X_test)

#visualisation
X_visual=X_val
Y_visual=y_val
Y_visual_pred =y_val_pred
pyplot.scatter(X_visual[:, 1], Y_visual, c="red")
pyplot.scatter(X_visual[:, 1], Y_visual_pred, c="blue")
lsit = merge_data(X_visual,Y_visual, Y_visual_pred)
print(pd.DataFrame(lsit))
print("\nRMSE:", 'train:', rmse(y_train_pred, y_train), ', valid:', rmse(y_val_pred, y_val),', test:', rmse(y_test_pred, y_test))
```

```
          0         1         2     3           4
0 -1.104083  0.828325 -1.362770  1.104  [1.3521944]
1  1.588802  0.828325 -1.362770  1.785  [1.8256444]
2 -1.104083  1.548607 -1.362770  1.134  [1.4246547]
3  1.588802  1.548607 -1.362770  1.719  [1.6987728]
4 -0.565506  1.548607 -1.362770  1.233  [1.4113814]
5 -0.026929  0.828325 -1.362770  1.328  [1.3580822]
6 -0.565506 -1.332522 -1.362770  1.199   [1.354088]
7 -0.565506 -0.612240 -1.362770  1.248  [1.1660714]
8 -0.026929  0.108042 -1.362770  1.479  [1.2648705]
9  1.588802  0.108042  0.733799  1.905  [1.7689041]

RMSE: train: 0.0629466157013901 , valid: 0.1661338885657776 , test: 0.1283509637416144
```



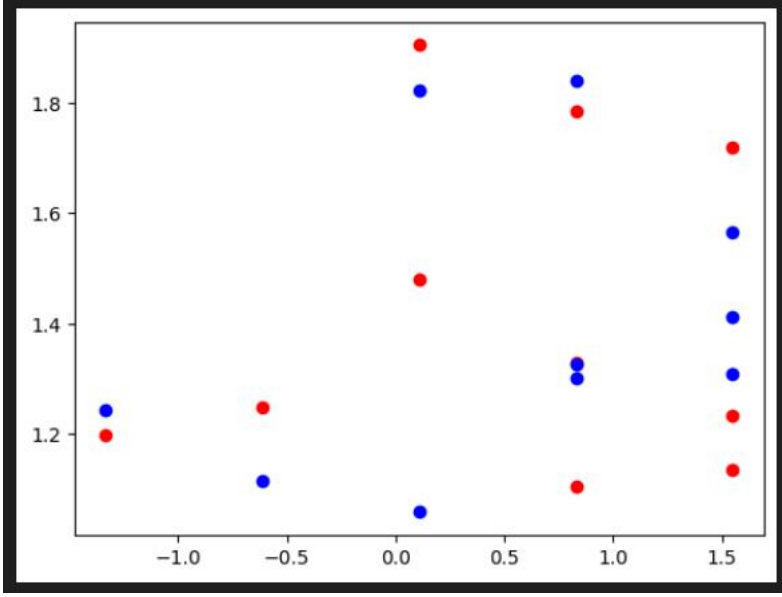
Komplizierte Regression mit höheren Epochwert:
Bei einem höheren Epochwert 300 bekommen wir einen sehr niedrigen RMSE Wert für die Trainingsdaten aber einen hohen Wert für die Validierungs- und Testdaten. In dem Fall hat sich unser Netzwerk auf die Trainingsdaten überspezialisiert, dass es fast keinen Fehler mehr macht aber bei neuen unbekannten Werten eine sehr hohe Fehlerquote hat. RMSE Unterschied is fast 0.230

```
large_nn = Sequential()
large_nn.add(InputLayer((3,)))
large_nn.add(Dense(256, 'relu'))
large_nn.add(Dense(128, 'relu'))
large_nn.add(Dense(64, 'relu'))
large_nn.add(Dense(32, 'relu'))
large_nn.add(Dense(1, 'linear'))

opt = Adam(learning_rate=.1)
cp = ModelCheckpoint('models/large_nn.keras', save_best_only=True)
large_nn.compile(optimizer=opt, loss='mse', metrics=[RootMeanSquaredError()])
large_nn.fit(x=X_train, y=y_train, validation_data=(X_val, y_val), callbacks=[cp], epochs=300)
```

✓ 1m 5.3s

	0	1	2	3	4
0	-1.104083	0.828325	-1.362770	1.104	[1.3267986]
1	1.588802	0.828325	-1.362770	1.785	[1.840675]
2	-1.104083	1.548607	-1.362770	1.134	[1.3092581]
3	1.588802	1.548607	-1.362770	1.719	[1.4115354]
4	-0.565506	1.548607	-1.362770	1.233	[1.5656205]
5	-0.026929	0.828325	-1.362770	1.328	[1.3012022]
6	-0.565506	-1.332522	-1.362770	1.199	[1.2438236]
7	-0.565506	-0.612240	-1.362770	1.248	[1.1141715]
8	-0.026929	0.108042	-1.362770	1.479	[1.0580808]
9	1.588802	0.108042	0.733799	1.905	[1.8220817]

RMSE: train: 0.009234192858428338 , valid: 0.22210605493248345 , test: 0.213652454929171

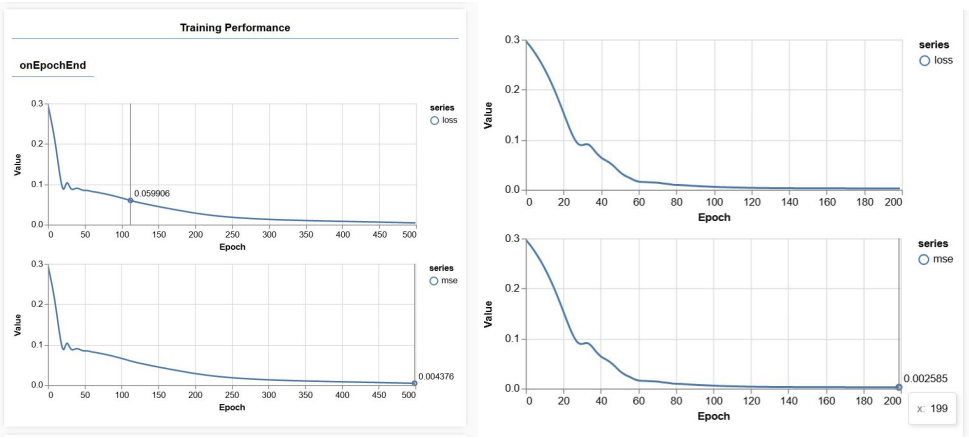


Javascript:

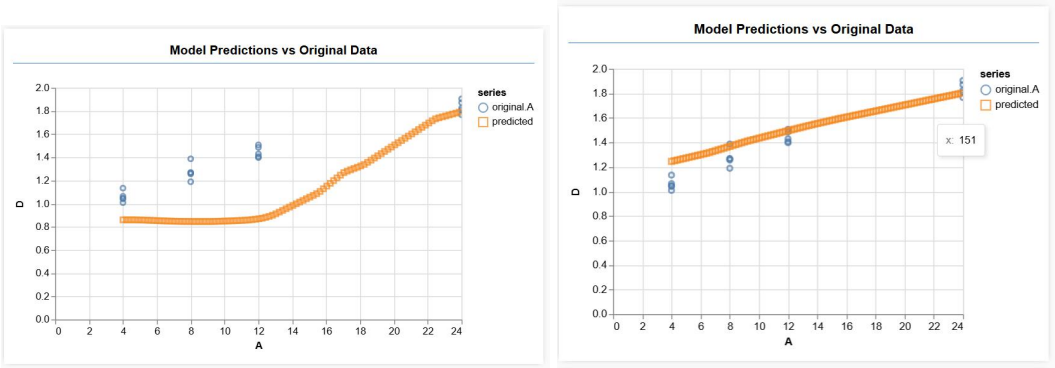
Für die Javascript Implementierung habe ich mich für eine komplizierte Regression entschieden und habe dies umgesetzt. Diese Regression hat 8 Relu Funktion aber keine Linear Funktion. Hier werden jedoch die Testdaten random generiert.

```
function createModel(rowCount) {
    // Create a sequential model
    const model = tf.sequential();

    // Add a single input layer
    model.add(
        tf.layers.dense({ inputShape: [rowCount], units: 1, useBias: true })
    );

    // activations
    model.add(
        tf.layers.dense({ inputShape: [256], units: 32, activation: "relu" })
    );
    model.add(
        tf.layers.dense({ inputShape: [128], units: 32, activation: "relu" })
    );
    model.add(
        tf.layers.dense({ inputShape: [64], units: 32, activation: "relu" })
    );
    model.add(
        tf.layers.dense({ inputShape: [32], units: 32, activation: "relu" })
    );
    model.add(
        tf.layers.dense({ inputShape: [16], units: 32, activation: "relu" })
    );
    model.add(
        tf.layers.dense({ inputShape: [8], units: 32, activation: "relu" })
    );
    model.add(
        tf.layers.dense({ inputShape: [4], units: 32, activation: "relu" })
    );
    model.add(
        tf.layers.dense({ inputShape: [2], units: 32, activation: "relu" })
    );
    // Add an output layer
    model.add(tf.layers.dense({ units: 1, useBias: true }));

    return model;
}
```

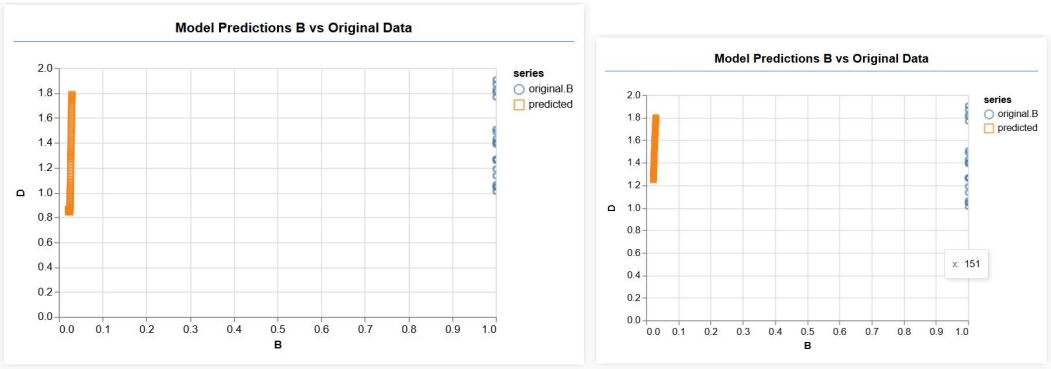
Bei einer Epoch von 500 ergibt sich einen MSE Wert von circa 0.004376 für die Trainingsdaten. Bei 200 ist der Wert 0.002585.



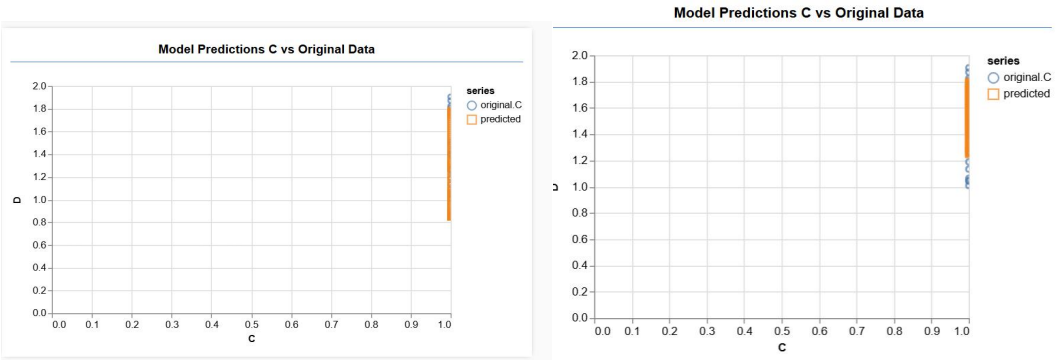
Zeitlicher Verlauf des MSE Werts je nach Epoch Wert. Epoch = 500 Links, 200 Rechts



Werte der erratenen D Werte bei zufällig generierte A Werten (orange) im Vergleich zu den erwartenen D Werten bei A Werte (blau). Epoch = 500 Links, 200 Rechts

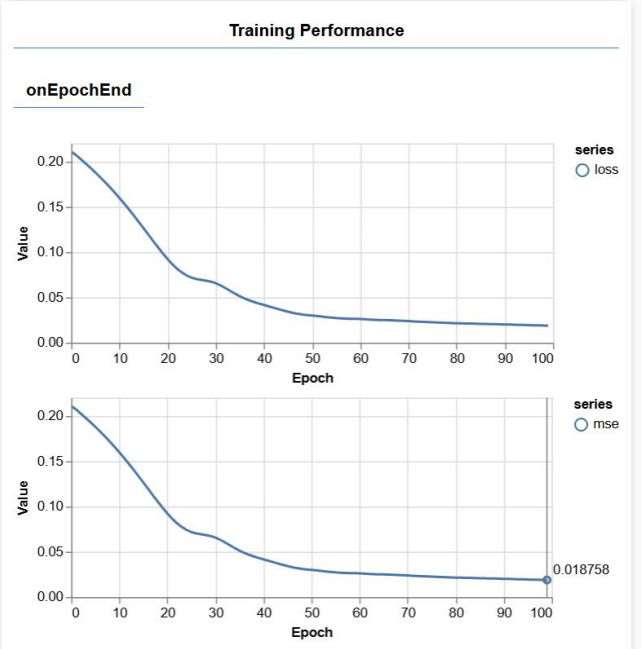


Werte der erratenen D Werte bei zufällig generierten B Werte (orange) im Vergleich zu den erwartenen D Werten bei B Werte (blau). Epoch = 500 Links, 200 Rechts



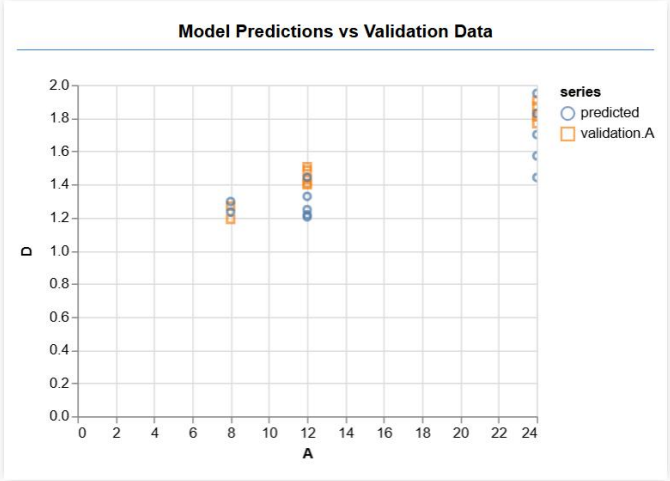
Werte der erratenen D Werte bei zufällig generierten C Werte (orange) im Vergleich zu den erwartenen D Werten bei C Werte (blau).Epoch = 500 Links, 200 Rechts

Was passiert wenn wir die Daten mit den Validierungsdaten vergleichen bei einer Epoch von 100?

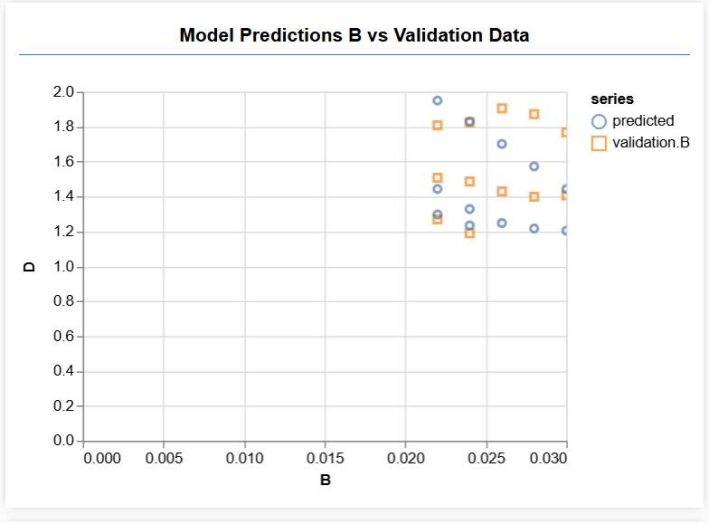


MSE wert von 0.0187 bei Epoch =100

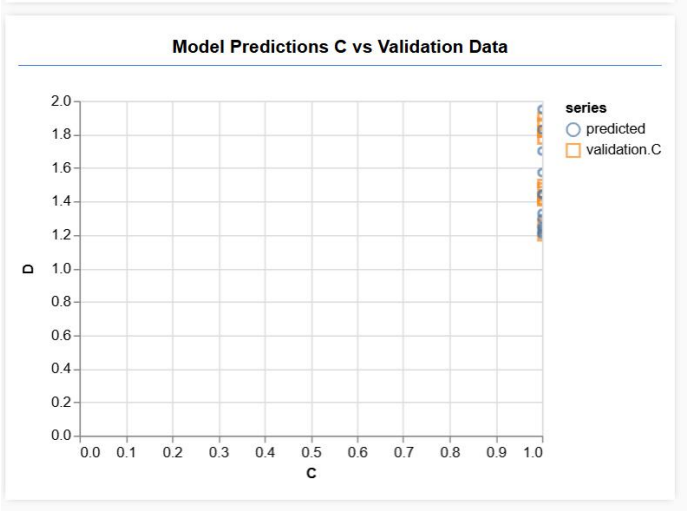
Erratene Werten im Vergleich zu den Validierungsdaten (Teil der Daten)



A vs D



B vs D



C vs D

Bei Epoch = 100 scheint die Daten gut zu passen!