Machine Learning I - Sheet 3

08.02.2020

Johannes Groß, 2036852, BSc Physik
Leonard Bleiziffer, 2038588, BSc Physik

1 Self organizing maps

A one dimensional self organizing map is used to solve the traveling salesman problem. In our example, we tried to find the shortest possible route for visiting all the beergardens of Regensburg. The GPS coordinates of the beergardens are used to place them in a two dimensional space.
The code provided in the github repository https://github.com/DiegoVicen/ntnu-som was used to initialize, train and plot the SOM. The code was only changed to be able to train the SOM on the beergarden data.
The used SOM has eight neurons per beergarden, the neighborhood was modeled with a Gaussian and the neighborhood radius as well as the learning rate decayed exponentially. Tuning the decay parameters of the radius and the learning rate has a great impact on both, the final solution and the number of iterations until convergence. Both decays should be chosen as fast as possible to speed up convergence, but when they are too rapid, the final solution will be far from optimal because too many neurons are dead too soon.

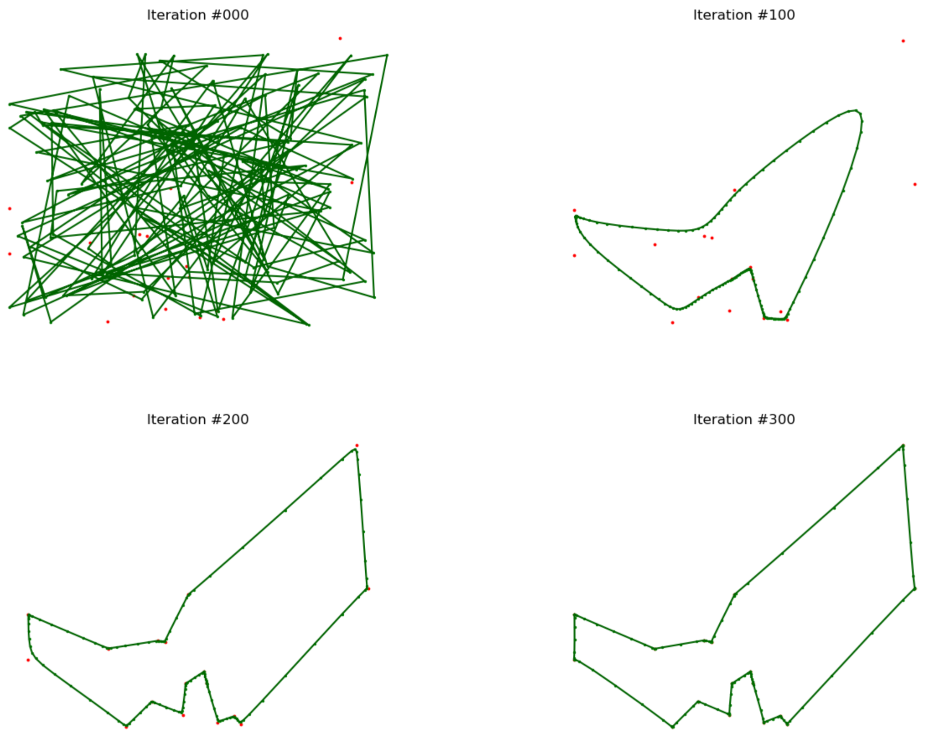

Abbildung 1.1: Snapshots of the solution from random initialization until convergence.

2 Multi layer perceptron

2.1 Regression

A multi layer perceptron (MLP) was used to approximate a non-linear function $f(x)$. Value pairs of x and $f(x)$ were provided for $x \in [0, 20]$. The MLP was trained with *tanh* and *logistic* as activation functions and with different numbers of hidden layers and hidden neurons.
In both cases the optimal architecture was 1 hidden layer with 2 neurons in it. This leads to an optimal result with the least complexity. However, in general the two activation functions react differently to different architectures. It can be observed that adding more hidden layers, while increasing the complexity, is not necessarily beneficial to the performance. Furthermore, we noticed that increasing the number of hidden layers demands an increase of neurons per layer in order to achieve good accuracy again. Of course, this further increases the compute time of the MLP. The complexity of the MLP is given by $O(n \cdot h_1 \cdot ... \cdot h_k \cdot i)$, where n is the number of samples, h_j is the number of neurons in the $j - th$ hidden layer and i is the number of iterations.
The notation for the architecture (i, j) means that the first hidden layer contains i neurons and the second hidden layer contains j neurons.

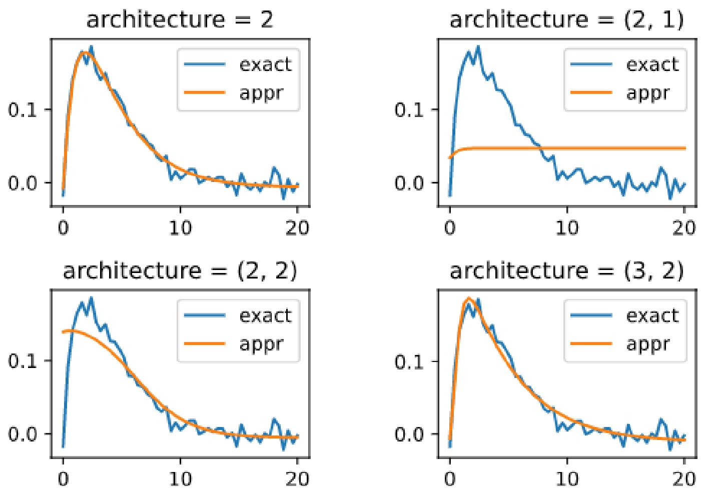

Abbildung 2.1: Plots for the estimation with the activation function *tanh*.

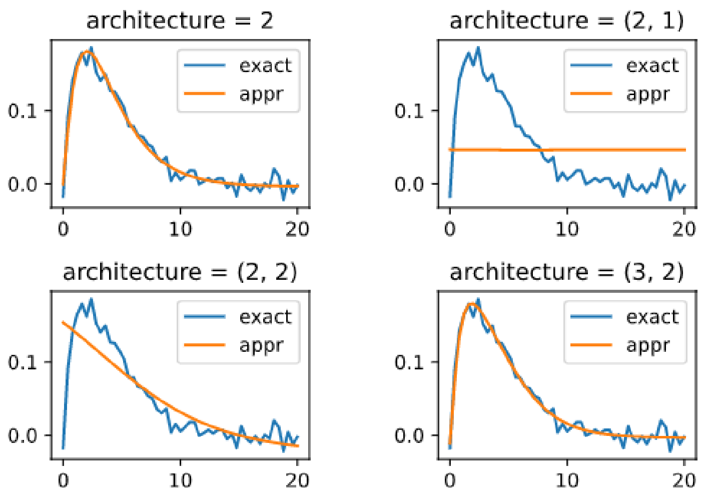

Abbildung 2.2: Plots for the estimation with the activation function *logistic*.

2.2 Classification

In this task, a MLP classifier was trained on the 'EMG Physical Action Data Set' from sheet 2 task 1. The architecture was optimized for the accuracy and compared to the support vector machine (SVM) and random forest (RF). The $logistic-function$ was used as activation as it performed better than $tanh$. The following tables show the performance of the MLP for different architectures, once with and once without standardizing the features by removing the mean and scaling to unit variance. The MLPs with the highest accuracies are colored:

	Accuracy	Time [s]
(50)	0.7806	12.93
(100)	0.7679	22.21
(200)	0.7811	38.11
(500)	0.7976	125.96
(50,50)	0.8327	47.86
(100,100)	0.8404	67.93
(200,200)	0.8345	337.75
(50,50,50)	0.8203	36.69
(100,100,100)	0.8302	95.60
(200,200,200)	0.8218	395.98

Tabelle 2.1: Results for different architectures on the non-standardized data set.

	Accuracy	Time [s]
(50)	0.8242	59.20
(100)	0.8423	132.14
(200)	0.8391	194.94
(500)	0.8476	375.64
(50,50)	0.8547	172.25
(100,100)	0.8512	266.16
(200,200)	0.8478	524.11
(50,50,50)	0.8505	217.53
(100,100,100)	0.8375	382.80
(200,200,200)	0.8413	626.55

Tabelle 2.2: Results for different architectures on the standardized data set.

Interestingly, the MLP classifier achieved a better accuracy on the standardized data, but only at the cost of a lot more compute time.

The following table compares the three different algorithms. RF achieves the highest accuracy and takes the least amount of time. Therefore, RF comes out as a clear winner on this data set. Between SVM and MLP one has to trade off between accuracy and computing time. The table shows the values for the green colored MLP, the one with the highest accuracy.

	Accuracy	Time [s]
SVM	0.7892	28.50
RF	0.8774	6.52
MLP	0.8547	172.25

MLP_Regression

February 8, 2021

```
[ ]: import numpy as np
     import pandas as pd
     from sklearn.model_selection import train_test_split
     from sklearn.neural_network import MLPRegressor
     import matplotlib.pyplot as plt
```

```
[ ]: #Save the training data in the nparray data
     data = np.loadtxt('data_function_approximation.txt')
```

```
[ ]: #Performs MLP regression for a given activation and architecture and plots the␣
      ↪results into an coordinate system
     def plot_mlp_regr(x, y, ax, activation, hidden_layer):
         #Initialize the model, perform the fit and make predictions
         reg = MLPRegressor(solver='lbfgs', activation=activation,␣
      ↪hidden_layer_sizes=hidden_layer, random_state=1)
         reg.fit(x.reshape(-1,1), y)
         y_pred = reg.predict(x.reshape(-1,1))

         #Plot the exact function and the estimate
         ax.set_title('architecture = ' + str(hidden_layer))
         ax.plot(x, y, label='exact')
         ax.plot(x, y_pred, label='appr')
         ax.legend()
```

```
[ ]: #Test MLP for different architectures and for different activation functions
     hidden_layer = [[(2), (2,1)], [(2,2), (3,2)]]
     activation = ['tanh', 'logistic']

     for activation in activation:
         fig, ax = plt.subplots(2,2)
         plt.subplots_adjust(hspace=0.5, wspace=0.5)
         print('\n' + str(activation) + ':')
         for i in range(2):
             for j in range(2):
                 plot_mlp_regr(data[:,0], data[:,1], ax[i,j], activation,␣
      ↪hidden_layer[i][j])
```

MLP_Classification

February 8, 2021

[ ]:
```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn import metrics
from sklearn.pipeline import make_pipeline
from sklearn.neural_network import MLPClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import glob
import time
```

[ ]:
```python
#Get and prepare all the data
aggr_files = []
normal_files = []
for i in range(1,5):
    path_aggr = r'EMG Physical Action Data Set/sub'+str(i)+'/Aggressive/txt'
    aggr_files.extend(glob.glob(os.path.join(path_aggr, "*.txt")))
    path_normal = r'EMG Physical Action Data Set/sub'+str(i)+'/Normal/txt'
    normal_files.extend(glob.glob(os.path.join(path_normal, "*.txt")))

df_from_each_aggr_file = (pd.read_table(f, names=list(range(8))) for f in␣
 ↪aggr_files)
aggr_df = pd.concat(df_from_each_aggr_file, ignore_index=True)

df_from_each_normal_file = (pd.read_table(f, names=list(range(8))) for f in␣
 ↪normal_files)
normal_df = pd.concat(df_from_each_normal_file, ignore_index=True)

aggr_df.insert(8, 'label', 0)
normal_df.insert(8, 'label', 1)

data = pd.concat([aggr_df,normal_df], ignore_index=True)
data = data.dropna()

feature_list = list(range(8))
```

```
[ ]: #Functions which calculates and prints different performance metrics
def MetricsOfReliability(clf, data_test_df, data_label_df, clf_predict_df):

    tn, fp, fn, tp = metrics.confusion_matrix(data_label_df, clf_predict_df).
 ↪ravel()

    accuracy = (tp + tn)/(tp + fp + tn + fn)
    print('accuracy = ' + str(accuracy))

    # sensitivity = tn/(tn + fp)
    # print('sensitivity = ' + str(sensitivity))

    # specificity = tp/(tp + fn)
    # print('specificity = ' + str(specificity))

    # metrics.plot_roc_curve(clf, data_test_df, data_label_df)
```

```
[ ]: #perform train-test-split
train, test = train_test_split(data, test_size=0.01, train_size=0.04)
```

```
[ ]: #Testing different architectures of the MLP and measuring accuracy and compute␣
 ↪time (Without standardizing the data)
hidden_layers = [(50), (100), (200), (500), (50,50), (100,100), (200,200),␣
 ↪(50,50,50), (100,100,100), (200,200,200)]
for hidden_layers in hidden_layers:
    tstart = time.time()
    mlp = MLPClassifier(activation='logistic',␣
 ↪hidden_layer_sizes=hidden_layers, max_iter=800, random_state=1)
    mlp.fit(train[feature_list], train['label'])
    mlp_predict = mlp.predict(test[feature_list])
    tend = time.time()

    print('\n' + str(hidden_layers) + ':')
    MetricsOfReliability(mlp, test[feature_list], test['label'], mlp_predict)
    print('time: ' + str(tend - tstart))
```

```
[ ]: #Testing different architectures of the MLP and measuring accuracy and compute␣
 ↪time (With standardizing the data)
scaler = StandardScaler()
train_tf = scaler.fit_transform(train[feature_list])
test_tf = scaler.transform(test[feature_list])

hidden_layers = [(50), (100), (200), (500), (50,50), (100,100), (200,200),␣
 ↪(50,50,50), (100,100,100), (200,200,200)]
for hidden_layers in hidden_layers:
    tstart = time.time()
    mlp = MLPClassifier(activation='logistic',␣
↪hidden_layer_sizes=hidden_layers, max_iter=800, random_state=1)
    mlp.fit(train_tf, train['label'])
    mlp_predict = mlp.predict(test_tf)
    tend = time.time()

    print('\n' + str(hidden_layers) + ':')
    MetricsOfReliability(mlp, test_tf, test['label'], mlp_predict)
    print('time: ' + str(tend - tstart))
```

[ ]:
```
#Measuring accuracy and compute time for the SVM with rbf kernel
tstart = time.time()
svm_rbf = make_pipeline(StandardScaler(), SVC(kernel='rbf'))
svm_rbf.fit(train[feature_list], train['label'])
svm_predict = svm_rbf.predict(test[feature_list])
tend = time.time()

MetricsOfReliability(svm_rbf, test[feature_list], test['label'], svm_predict)
print('time: ' + str(tend - tstart))
```

[ ]:
```
#Measuring accuracy and compute time for the RF
tstart = time.time()
rf = RandomForestClassifier()
rf.fit(train[feature_list], train['label'])
rf_predict = rf.predict(test[feature_list])
tend = time.time()

MetricsOfReliability(rf, test[feature_list], test['label'], rf_predict)
print('time: ' + str(tend - tstart))
```