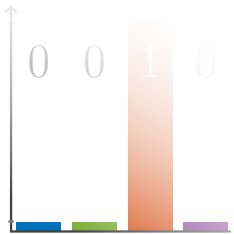
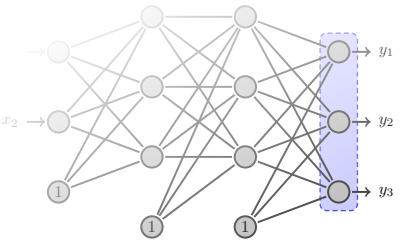
Klassifikation mit Neuronalen Netzen

Prof. Dr. Jörg Frochte

Maschinelles Lernen





Classification Error, Accuracy und Mean Squared Error

- Beurteilung der Qualität einer Klassifikation erfolgt auf unterschiedliche Weise.
- Wir haben u.a. den **Classification Error** berechnet. Dieser ist definiert als:

$$\text{Classification Error} = \frac{\text{Anzahl der Fehlklassifizierungen}}{\text{Anzahl der Fälle}}$$

- Die Genauigkeit bzw. **Accuracy** ist entsprechend

$$\text{Accuracy} = 1 - \text{Classification Error}$$

- Dies sind Maße, die wir nach der Optimierung der Gewichte in einem neuronalen Netzen bestimmen.
- Für die optimale Festlegung der Gewichte haben wir bisher nur den **Mean Squared Error** als Kriterium verwendet.
- Dieser funktioniert auch für die Klassifikation oft, ist jedoch dort nicht die erste Wahl und wir lernen nun eine i.d.R. bessere Alternative kennen.

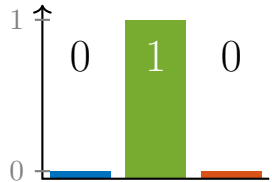
Sigmoid vs. Softmax

- Für dieses neue Maß benötigen wir u.a. eine Veränderung im Output.
- Bei Klassifikationen kann im Output-Layer eine Sigmoid-Aktivierung verwendet werden.
- Die Summe ist nicht 1 und somit entsprechen Sigmoid-Outputs nicht Wahrscheinlichkeiten.
- Um dies sicherzustellen, benutzen wir die **Softmax**-Funktion:

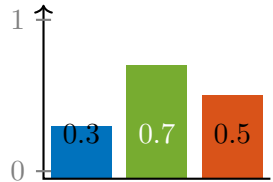
$$\sigma_j(z) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ für } j = 1..n \tag{1}$$

- Praktisch nutzt man solche Softmax-Funktionen direkt als Aktivierungsfunktion in Umsetzungen für neuronale Netze, wie z.B. Keras/Tensorflow.
- Wir führen jedoch nun zunächst das Gedankenexperiment durch, was passieren würde, wenn wir diese Funktion einfach auf den Output eines bereits mit Sigmoid-Funktionen trainierten ANN anwenden.

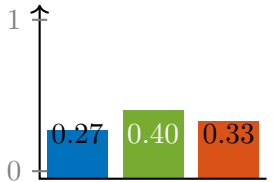
Sigmoid vs. Softmax



Trainingsdaten



Netz mit Sigmoid-Output



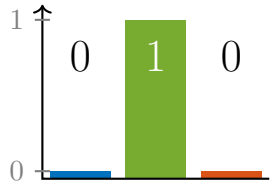
nach Anwendung von Softmax

- Nun haben wir etwas das sich wie eine Wahrscheinlichkeit für eine Klasse verhält.
- Hierauf kann man die Cross-Entropy als Fehlerfunktion einführe:

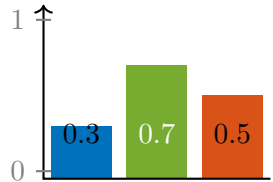
$$D(y, y_p) = -\sum_i y_i \log(y_{p_i}) \tag{2}$$

- Welche Basis im Logarithmus hier verwendet wird ist nicht erheblich, wir nehmen normalerweise den logarithmus naturalis.

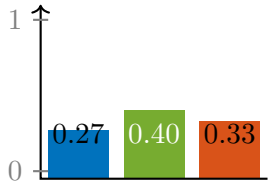
Softmax und Cross-Entropy



Trainingsdaten



Netz mit Sigmoid-Output



nach Anwendung von Softmax

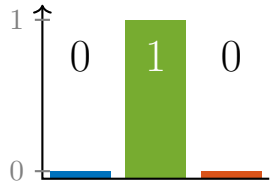
- Worauf man hingegen achten muss ist, dass die Cross-Entropy nicht symmetrisch ist:

$$D(y_p, y) \neq D(y, y_p)$$

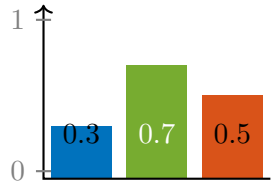
- Der Logarithmus von Null exisitiert nicht, daher wird eine untere Schranke $\varepsilon > 0$ verwendet.
- Berechnen wir die Cross-Entropy für das Beispiel oben mit dem natürlichen Logarithmus:

$$D(y, y_p) = -\sum_i y_i \log(y_{p_i}) = -(1 \cdot \log(0.40) + 0 \cdot \log(0.27) + 0 \cdot \log(0.34)) \approx 0.91$$

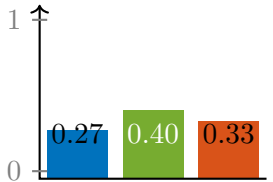
Softmax und Cross-Entropy



Trainingsdaten



Netz mit Sigmoid-Output



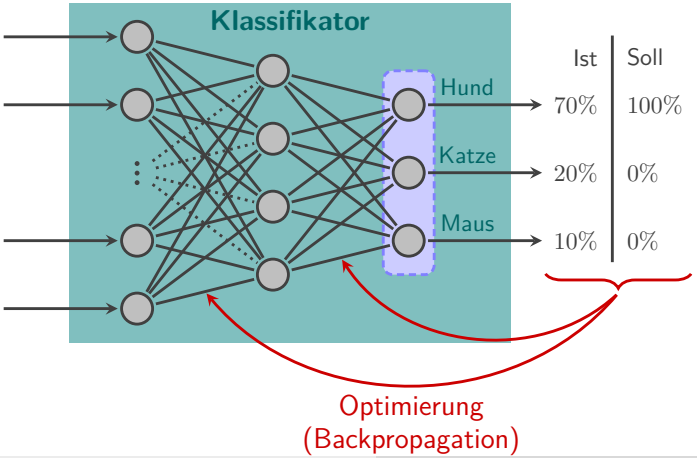
nach Anwendung von Softmax

- Da der Logarithmus von 1 immer 0 ist und dazu auch noch streng monoton steigend, haben wir hier ein sinnvolles Maß, nach dem wir optimieren können.
- Ziel: Der Null möglichst nah kommen.
- Die Fehlerfunktion $J(W)$ zur Optimierung der Gewichte entsteht nun, indem wir die Cross-Entropy über alle N Beispiele berechnen und anschließend mitteln:

$$J(w) = \frac{1}{N} \sum_{n=1}^{N} D(y^{(n)}, y_p^{(n)}) \tag{3}$$

Softmax Darstellung am Netz

- In Keras/Tensorflow wird der Softmax nicht auf einen Output aufgesetzt sondern ist selbst eine Aktivierungsfunktion.



Mean Squared Error vs. Cross-Entropy-Error

Mean Squared Error (üblich für Regressionsprobleme)

$$J(w) = \frac{1}{N} \sum_{n=1}^{N} \sum_{i} (y_i^{(n)} - y_{p_i}^{(n)})^2$$

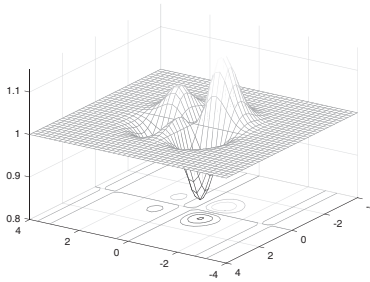
Cross-Entropy-Error (üblich für Klassifikation; nur mit Wahrscheinlichkeiten)

$$J(w) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{i} y_i^{(n)} \log(y_{p_i}^{(n)})$$

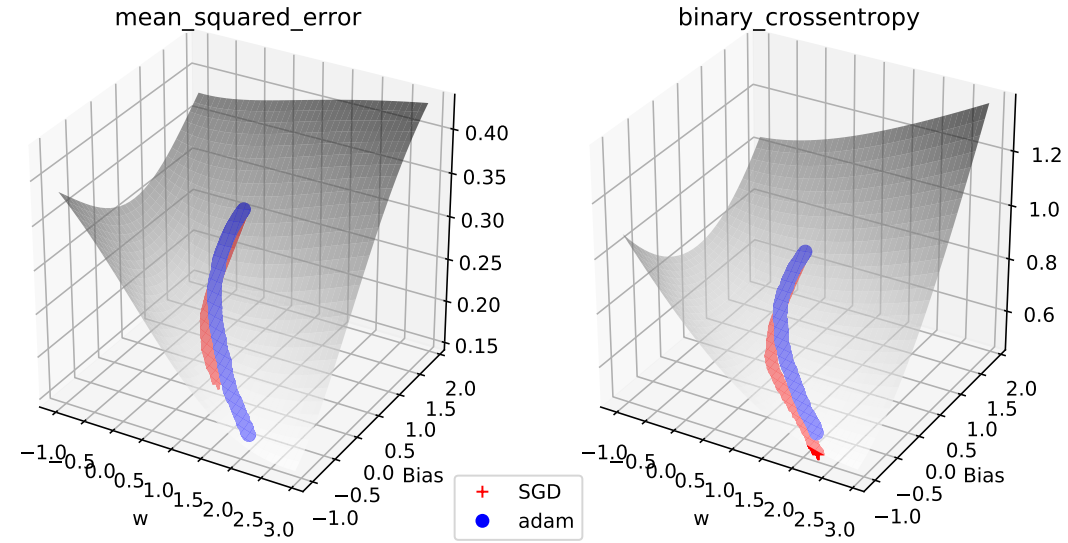
Trainingsmenge

Vorhergesagt

Klassen	2 Klassen	n Klassen ($n \geq 2$)
Outputs	1 Output mit Sigmoid	n Outputs mit Softmax
Beispiel	$[0]$ oder $[1]$	$[1,0,0]$, $[0,1,0]$, $[0,0,1]$
Loss	`binary_crossentropy`	`categorical_crossentropy`



Unterschiede am Beispiel



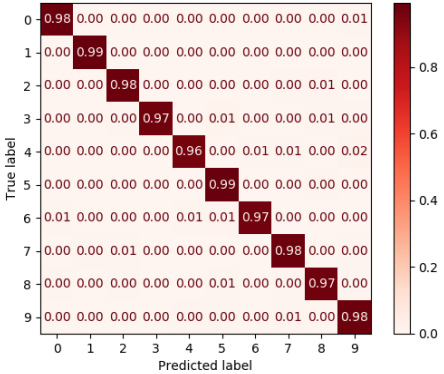
```
1  from tensorflow.keras.models import Sequential
2  from tensorflow.keras.layers import Dense
3  from tensorflow.keras.utils import to_categorical
4  from tensorflow.keras.datasets import mnist
5
6  (XTrain, yTrain), (XTest, yTest) = mnist.load_data()
7
8  XTrain = XTrain.reshape(60000, 784)
9  XTest = XTest.reshape(10000, 784)
10 XTrain = XTrain/255
11 XTest  = XTest/255
12 YTrain = to_categorical(yTrain, 10)
13 YTest  = to_categorical(yTest, 10)
14
15 myANN = Sequential()
16 myANN.add(Dense(80,input_dim=784,activation='relu'))
17 myANN.add(Dense(40,activation='relu'))
18 myANN.add(Dense(10,activation='softmax'))
```

```
19 myANN.compile(loss='categorical_crossentropy', optimizer='adam',
20               metrics=['accuracy'])
21 myANN.fit(XTrain, YTrain, batch_size=24, epochs=10, verbose=True)
22
23 score = myANN.evaluate(XTest, YTest, verbose=False)
24 print('Test loss value :', score[0]) # 0.0991
25 print('Test metric value [accuracy]:', score[1]) # 0.9797
26 myANN.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 80)                62800
_________________________________________________________________
dense_4 (Dense)              (None, 40)                3240
_________________________________________________________________
dense_5 (Dense)              (None, 10)                410
=================================================================
Total params: 66,450
Trainable params: 66,450
Non-trainable params: 0
```

Konfusionsmatrix

```
27
28 from sklearn.metrics import confusion_matrix
29 from sklearn.metrics import ConfusionMatrixDisplay
30 import numpy as np
31 yP = np.argmax(myANN.predict(XTest),axis=1)
32 cm=confusion_matrix(yTest, yP, normalize='true')
33 labels=['0','1','2','3','4','5','6','7','8','9']
34 cmd=ConfusionMatrixDisplay(cm,
35                            display_labels=labels)
36 cmd.plot(cmap='Reds', values_format='.2f')
```



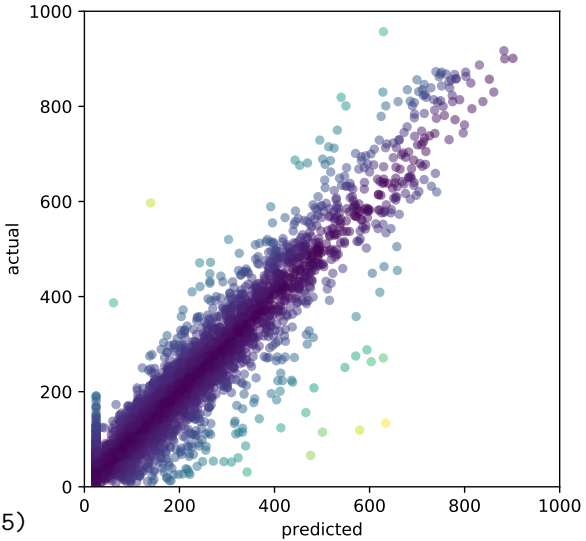
- Wie schon beim Bayes-Klassifikator erwähnt, ist die Konfusionsmatrix ein gutes Mittel um die Qualität einer Klassifikation genauer zu betrachten.
- Noch kurz ein Exkurs zu einem weiteren Ansatz für die Regression.

Scatter-Plot

- Bei Regressionsproblemen kann man keine Konfusionsmatrix erstellen.
- Aber um zu sehen, wie die Vorhersagen bezogen auf die tatsächlichen Zielwerte verteilt sind, können wir beide Größen im Scatter-Plot gegeneinander auftragen.
- Dies Hilft bei der Bewertung des Modells.
- Im Fall rechts (Bike-Sharing mittels ANN) fällt auf, dass ungewöhnlich oft ein sehr kleiner Wert vorhergesagt wird (knapp 25 Fahrräder). Das könnte man untersuchen.

Code:

```
absErr = np.abs(yp - ytest)
plt.scatter(yp, ytest, c=absErr, alpha=0.5)
```



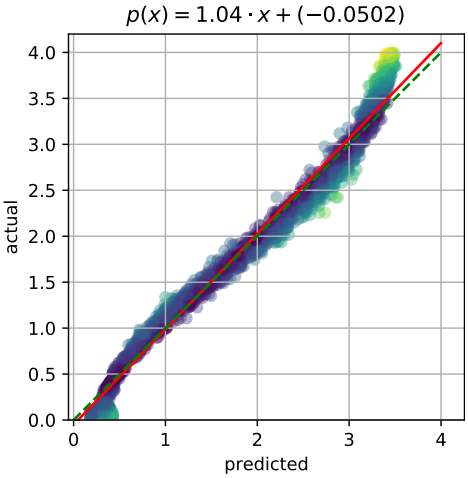
Scatter-Plot mit Regressionsgerade

- Zusätzlich kann man noch eine Regressionsgerade über ein lineares Ausgleichsproblem bestimmen:

$$\begin{pmatrix} \hat{y}_1 & 1 \\ \vdots & \vdots \\ \hat{y}_n & 1 \end{pmatrix} \cdot \begin{pmatrix} m \\ b \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

- Optimalerweise wäre $m = 1$ und $b = 0$.
- Code (zusätzlich zum Scatter-Plot):

```
m, b = np.polyfit(yp, ytest, deg=1)
plt.plot([0, 4], [b, 4 * m + b], 'r')
plt.plot([0, 4], [0, 4], 'g--')
```



Boston-Housing Dataset: Train vs. Test