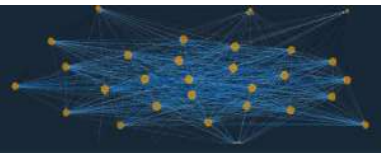
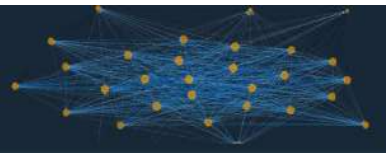


Deep Learning: Neuronale Netze mit Keras

Keras Layers Klasse



```
tf.keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
**kwargs= [name, dtype, trainable]
```

Dense-Schicht: Alle Neuronen sind mit allen Merkmalen verknüpft

Units: Anzahl der Neuronen der Schicht

Activation: Aktivierungsfunktion, None: Linear

Use_bias: Sollen die Neuronen einen Bias haben?

Kernel_initilaizer: Initialisierung der Gewichtungen

Bias_initilaizer: Initialisierung der Bias-Werte

Kerel_regularizer: Regularisierung der Gewichtungen um Overfitting zu vermeiden

Bias_regularizer: Regularisierung der Bias-Werte

Activity_regularizer: Regularisierung nach der Aktivierungsfunktion

Kernel_constraints: Limits für die Gewichtungen

Bias_constraints: Limits für die Bias-Werte

https://keras.io/api/layers/core_layers/dense/

Sequential-Methode

```
from tensorflow import keras
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import r2_score

(x_train, y_train), (x_val, y_val) = keras.datasets.california_housing.load_data()

scaler_x = MinMaxScaler()
scaler_x.fit(x_train)
x_train = scaler_x.transform(x_train)
x_val = scaler_x.transform(x_val)

y_max = y_train.max()
y_train = y_train/y_max
y_val = y_val/y_max

# Sequential
model1 = keras.models.Sequential()
model1.add(keras.layers.Input(x_train.shape[1:]))
model1.add(keras.layers.Dense(300, activation="relu"))  # hidden1
model1.add(keras.layers.Dense(300, activation="relu")) # hidden2
model1.add(keras.layers.Dense(1)) # outputlayer
model1.summary()

model1.compile(loss="mse", optimizer=keras.optimizers.SGD(learning_rate=0.001))

model1.fit(x_train, y_train, epochs=500, validation_data=(x_val, y_val))

print('Training r2_score', r2_score(y_train,model1.predict(x_train)))
print('Validation r2_score', r2_score(y_val,model1.predict(x_val)))
```

keras.models.Sequential:

- Definiere das neuronale Netz als Abfolge von Schichten
- Der Output einer Schicht ist automatisch der Input für die nachfolgende Schicht
- Schichten sind Instanzen von keras.layers

👍 Einfach anzulegen

👍 Übersichtlich

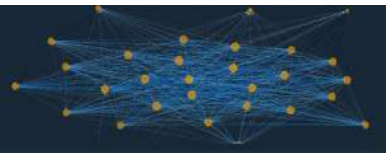
👎 Geringer Einblick in interne Berechnung

👎 Wenig anpassungsfähig

👎 Keine Verzweigungen

👎 Starrer Ablauf beim Training

Functional-Methode



```
from tensorflow import keras
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import r2_score

(x_train, y_train), (x_val, y_val) = keras.datasets.california_housing.load_data()

scaler_x = MinMaxScaler()
scaler_x.fit(x_train)
x_train = scaler_x.transform(x_train)
x_val = scaler_x.transform(x_val)

y_max = y_train.max()
y_train = y_train/y_max
y_val = y_val/y_max

# Functional
model_input = keras.layers.Input(x_train.shape[1:])
hidden1 = keras.layers.Dense(300, activation="relu")(model_input)
hidden2 = keras.layers.Dense(300, activation="relu")(hidden1)
output1 = keras.layers.Dense(1)(hidden2)
model2 = keras.models.Model(model_input, output1)
model2.summary()

model2.compile(loss="mse",optimizer=keras.optimizers.SGD(learning_rate=0.001))

model2.fit(x_train, y_train, epochs=500, validation_data=(x_val,y_val))

print('Training r2_score', r2_score(y_train,model2.predict(x_train)))
print('Validation r2_score', r2_score(y_val,model2.predict(x_val)))
```

Functional-Methode:

- Definiere das neuronale Netz als Abfolge von Gleichungen
- Der Output muss als Input für die nächste Schicht 'durchgefädelt' werden
- Schichten sind Instanzen von keras.layers

👍 Einfach anzulegen

👍 Übersichtlich

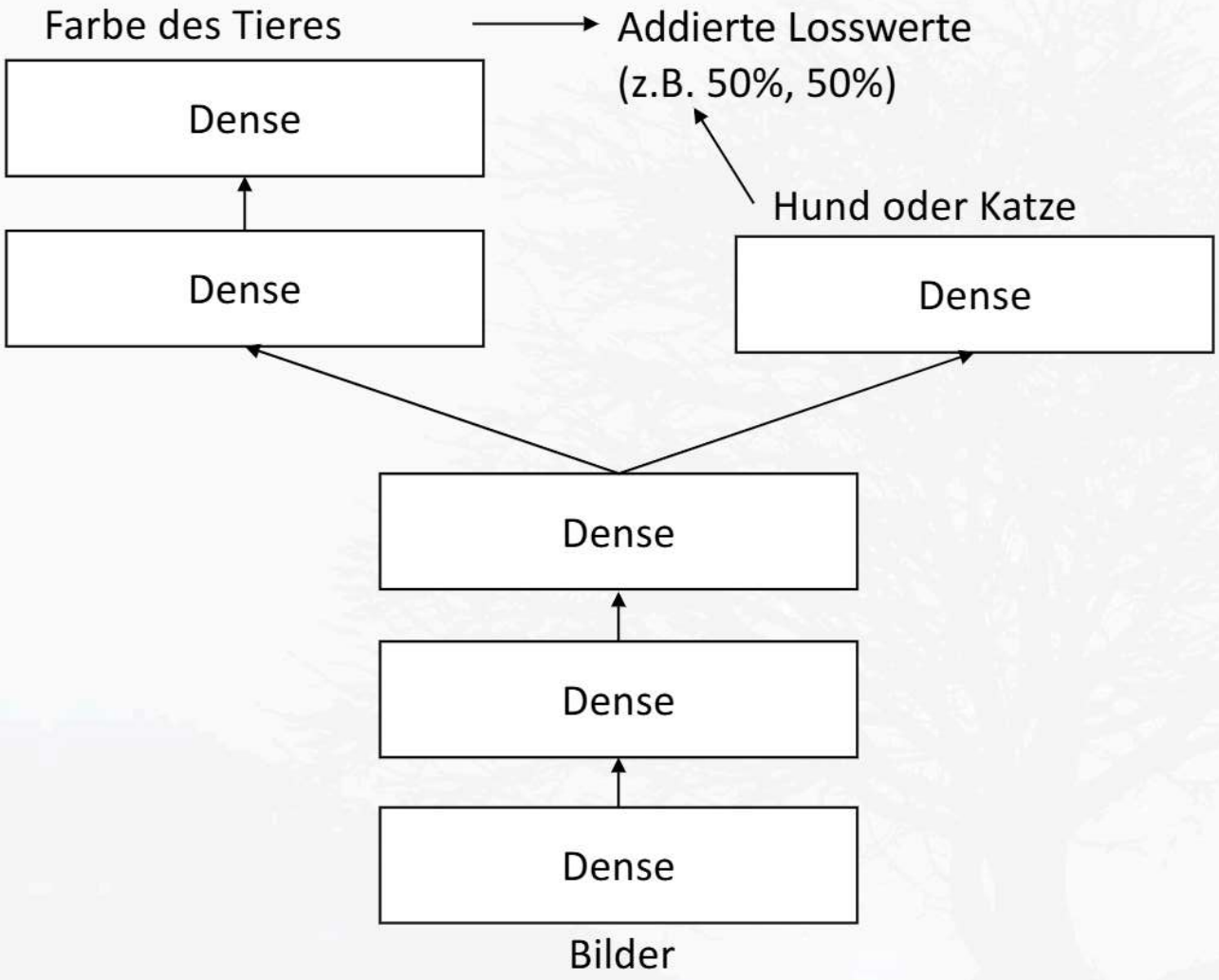
👍 Guter Einblick in interne Berechnung

👍 Beliebige Verzweigungen

👍 Gute Anpassungsfähigkeit

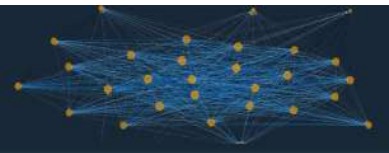
👎 Starrer Ablauf beim Training

Verzweigungen



- Verzweigungen sind nützlich, wenn zwischen mehreren Labels eine Abhängigkeit besteht
- Gemeinsame Schichten können die wesentlichen Strukturen analysieren
- Verzweigte Äste spezialisieren sich auf die jeweiligen Labels
- Für jeden Output wird eine Loss-Funktion definiert, beide Losses werden anteilig addiert
- Es lassen sich so auch Regression und Klassifikation kombinieren

Subclassing-Methode



.
.
.

```
# Subclassing
class SubClassModel(keras.models.Model):
    def __init__(self):
        super().__init__()
        self.layer1 = keras.layers.Dense(300, activation="relu")
        self.layer2 = keras.layers.Dense(300, activation="relu")
        self.output_layer = keras.layers.Dense(1)

    def call(self, input):
        hidden1 = self.layer1(input)
        hidden2 = self.layer2(hidden1)
        model_output = self.output_layer(hidden2)
        return model_output

model3 = SubClassModel()
```

```
model3.compile(loss="mse", optimizer=keras.optimizers.SGD(learning_rate=0.001))
model3.fit(x_train, y_train, epochs=500, validation_data=(x_val, y_val))
```

.
.

Subclassing-Methode:

- Definiere das neuronale Netz als Klasse mit den Schichten als Attribute
- Definiere eine Funktion *call* für die Berechnung des Outputs
- Output/Input muss durchgefädelt werden

👍 Eigene Layertypen möglich

👍 Größte Flexibilität

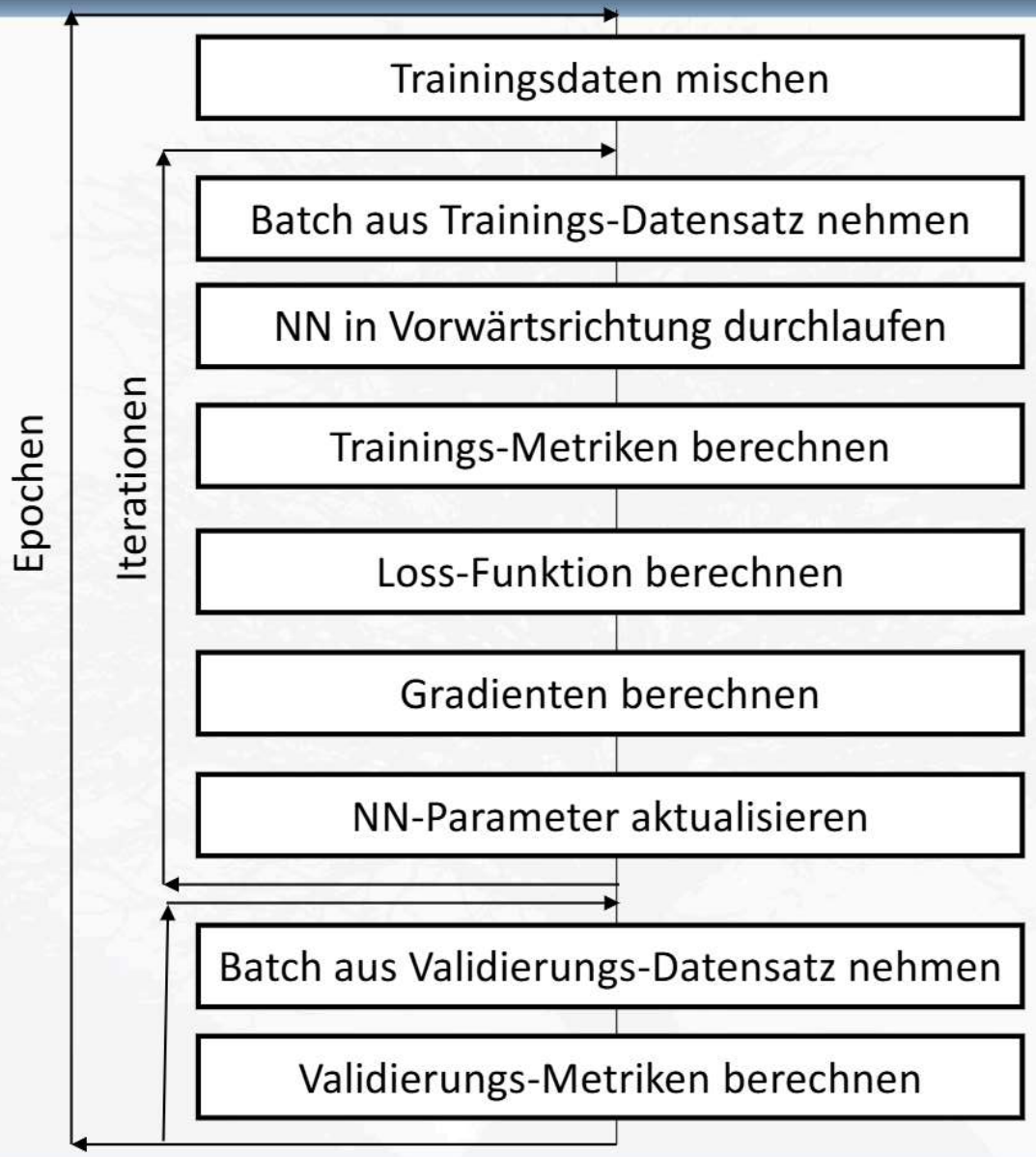
👍 Bedingte Verzweigungen

👍 Interne Schleifen

👎 Eher unübersichtlich

👎 Von außen, wenig Einblick in Berechnung

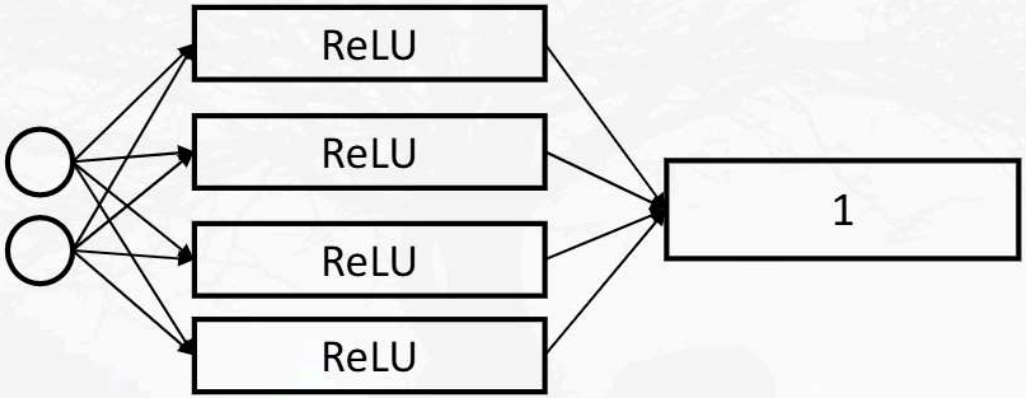
Fitprozess



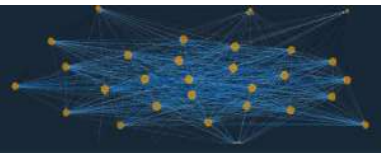
https://keras.io/api/models/model_training_apis/

Typische Architektur für Regression

- Ein einzelnes Output-Neuron
- Hiddenlayers sind Dense Layer (fully-connected)
- Anzahl der Neuronen pro Schicht wird zunächst größer, dann wieder kleiner
- ReLU, oder Leaky-ReLU als Aktivierungsfunktion der Hiddenlayers
- Keine Aktivierungsfunktion für das Output-Neuron
- Mean-Squared-Error als Loss-Funktion



Regressionsmetriken



Mittlerer absoluter Fehler (Optimum: 0):

```
import tensorflow as tf
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]
mae = tf.keras.losses.MeanAbsoluteError()
mae_score = mae(y_true, y_pred).numpy()
```

https://keras.io/api/metrics/regression_metrics/

$$mae\_score = \frac{1}{n}\sum_{i=1..n}\left|y_i - y_{pred,i}\right|$$

Bestimmtheitsmaß (Optimum: 1):

```
from sklearn.metrics import r2_score
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]
r2_score(y_true, y_pred)
```

https://scikit-learn.org/stable/modules/model_evaluation.html#r2-score

$$R^2 = 1 - \frac{\sum_{i=1..n}\left[y_i - y_{pred,i}\right]^2}{\sum_{i=1..n}\left[y_i - \bar{y}\right]^2}$$

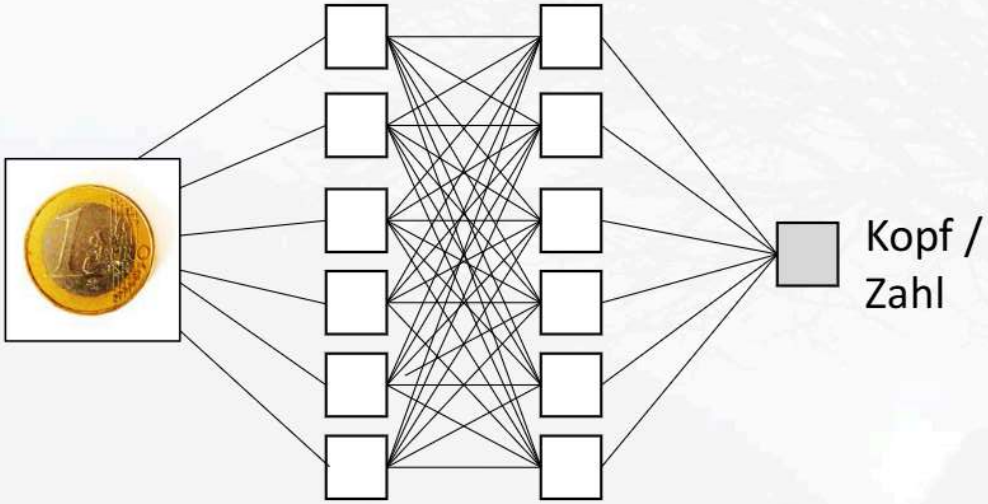
$\bar{y}$ = Mittelwert von y

Typische Architektur für Klassifikation

Binäre Klassifikation:

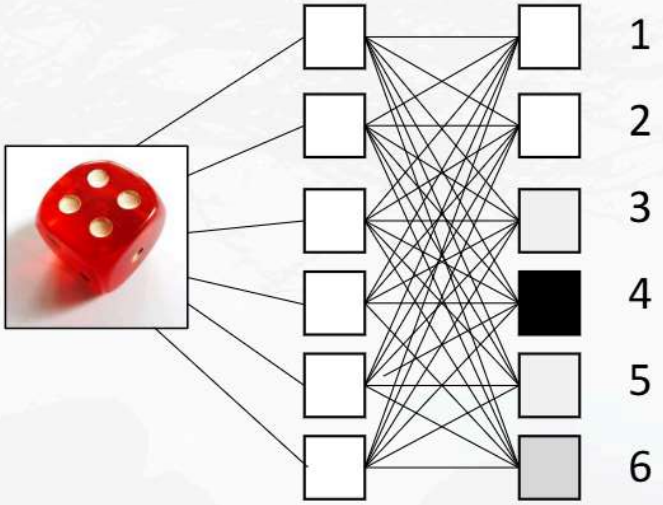
- Die Labels haben die Werte 0 und 1
- Ein einzelnes Output-Neuron
- Sigmoid als Aktivierungsfunktion für das Output-Neuron
- Binary_Crossentropy als Loss-Funktion

$$loss = -\frac{1}{n}\sum_{i=1..n} y_i \ln(y_{pred,i}) + (1 - y_i) \ln(1 - y_{pred,i})$$

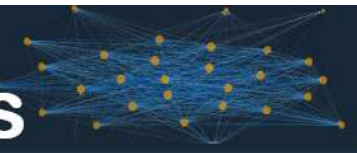


Multiclass-Klassifikation

- Jedes Label eigene Spalte in y (one-hot-encoded)
- y = [2, 1, 0] → y =[[0, 0, 1], [0, 1, 0], [1, 0, 0]]
- Ein Output-Neuron für jede Klasse
- Softmax als Aktivierungsfunktion für die Output-Neuronen
- Categorical_Crossentropy als Loss-Funktion



Softmax und Categorical-Cross-Entropy-Loss



```
import tensorflow as tf

y=tf.Variable([0., 0., 1.])
y_sum=tf.Variable([10., 10., 0.1])

layer = tf.keras.layers.Softmax()
y_softmax=layer(y_sum).numpy()

cce = tf.keras.losses.CategoricalCrossentropy()
loss=cce(y, y_softmax).numpy()
```

Softmax-Aktivierung

$$y_{pred}^{(i)} = \frac{\exp\left(y_{sum}^{(i)}\right)}{\sum_{j=1..D} \exp(y_{sum}^{(j)})}$$

Categorical_Cross_Entropy-Loss

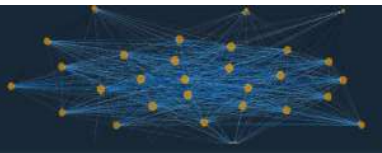
$$loss = -\frac{1}{n} \sum_{i=1..n} \sum_{j=1..D} y_i^{(j)} \ln\left(y_{pred,i}^{(j)}\right)$$

Sparse_Categorical_Cross-Entropy-Loss

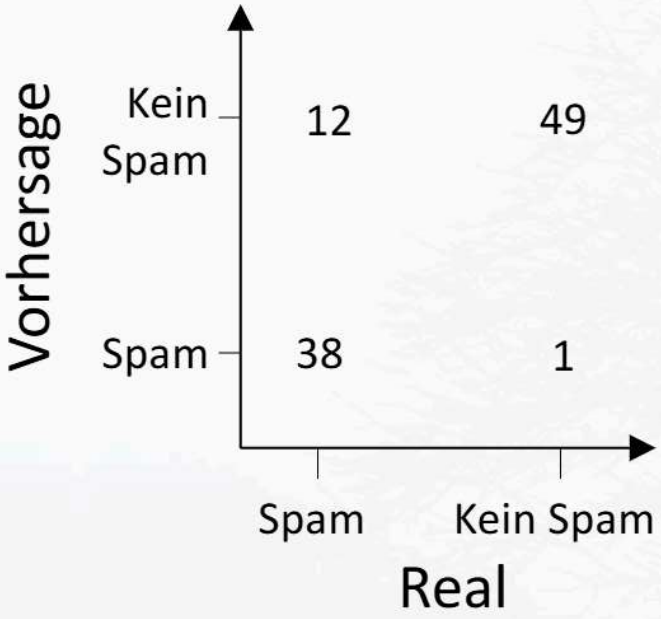
$$loss = -\frac{1}{n} \sum_{i=1..n} \ln\left(y_{pred,i}[y_i]\right)$$

- Softmax-Aktivierung bewirkt, dass y_{pred} in Summe auf 1 normalisiert ist
- Jedes Neuron gibt dann die Wahrscheinlichkeit an, dass die jeweilige Klasse vorliegt
- y und y_{pred} sind kategorisch (z.B. y_1=[0, 0, 1], y_{1pred}=[0.274, 0.274, 0.452])
- Categorical-Cross-Entropy-Loss wirkt nur auf die Elemente bei denen y==1

Einfache Klassifikation: Spam, Ja / Nein?



100 E-Mails:
50 Spam, 50 Kein Spam



- Richtige Vorhersagen = (38 + 49) / 100
- Falsche Vorhersagen = (12 + 1) / 100

Spamfilter darf keine persönlichen Emails aussortieren

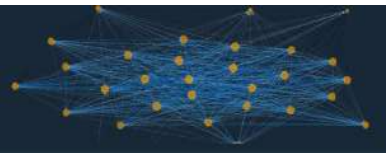
⇨ Genauere Unterscheidung nötig:

- Falsch Negativ (FN) = 12
- Richtig Negativ (RN) = 49
- Richtig Positiv (RP) = 38
- Falsch Positiv (FP) = 1

⇨ Bei Spamfilter: FP sollte 0 sein

Spam nur markieren, wenn Wahrscheinlichkeit >80 %
Dafür höheres FN in Kauf nehmen

Metriken bei binärer Klassifikation



$$\text{Falsch}-\text{Positiv}-\text{Rate} = \frac{\text{FP}}{\text{FP}+\text{RN}} = \frac{1}{1+49} \qquad \text{Spezifität} = \frac{\text{RN}}{\text{RN}+\text{FP}} = \frac{49}{49+1}$$

$$\text{Sensitivität} = \frac{\text{RP}}{\text{RP}+\text{FN}} = \frac{38}{38+12} \qquad \text{Falsch}-\text{Negativ}-\text{Rate} = \frac{\text{FN}}{\text{FN}+\text{RP}} = \frac{12}{12+38}$$

$$\text{Korrektklassifikationsrate} = \frac{\text{RP}+\text{RN}}{\text{RP}+\text{FP}+\text{FN}+\text{RN}} = \frac{38+49}{38+1+12+49}$$

$$\text{Falschklassifikationsrate} = \frac{\text{FP}+\text{FN}}{\text{RP}+\text{FP}+\text{FN}+\text{RN}} = \frac{1+12}{38+1+12+49}$$

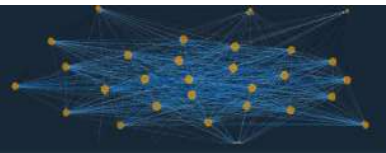
'accuracy' # Korrektklassifikationsrate = $\frac{\text{RP}+\text{RN}}{\text{RP}+\text{FP}+\text{FN}+\text{RN}}$

'recall' # Sensitivität = $\frac{\text{RP}}{\text{RP}+\text{FN}}$

'precision' # Genauigkeit = $\frac{\text{RP}}{\text{RP}+\text{FP}}$

'f1' # F-Maß = $\frac{2\times(\text{Genauigkeit}\times\text{Sensitivität})}{(\text{Genauigkeit}+\text{Sensitiviät})} = \frac{2\text{RP}}{2\text{RP}+\text{FN}+\text{FP}}$

Unbalancierte Labels: Sampling Methoden



Es liegen drei Klassen vor: Von Label=0 gibt es 5567, von Label=1: 354 und von Label=2: 3224 Datenpunkte

Aufgrund des Ungleichgewichts, wird das neuronale Netz Label=0 bevorzugt prognostizieren. Der Datensatz stellt eine Stichprobe aus der realen Verteilung dar

- Keine Anpassung notwendig, der Datensatz spiegelt die Realität wider: Z. B. die Labels entsprechen verkauften Produkten eines Supermarkts: Äpfel, Birnen und Bananen. Es werden am meisten Äpfel, wenige Birnen und relativ viele Bananen verkauft. Es ist sinnvoll, wenn das neuronale Netz diese Verteilung auch in der Prognose widerspiegelt
- Anpassung notwendig, der Datensatz spiegelt die Realität inkorrekt wider: Z. B. die Labels gelten für Hunde-, Katzen- und Pferdebilder welche von einer Webseite geladen wurden. Das neuronale Netz soll in diesem Fall primär eine Gleichverteilung der Labels zugrunde legen. Wenn wir das Bild nicht sehen würden, erwarteten wir eine Wahrscheinlichkeit von je 1/3 für die einzelnen Klassen
  - Undersampling: Von jeder Klasse werden beliebig nur 354 Bilder für das Training ausgewählt
  - Oversampling: Katzen- und Pferdebilder werden im Training mehrfach verwendet
  - Gewichtung der Klassen: Im Training können die Klassen mit weniger Datenpunkten stärker gewichtet werden. Z. B. class_weight={0:1, 1:5567/354, 2:5567/3224}

Kalibrierung der Wahrscheinlichkeiten

Die Prognose eines Klassifikators gibt die Wahrscheinlichkeit an, ob eine Klasse vorliegt

Bei der Multiclass-Klassifikation gibt es ein Label für jede Klasse, die in Summe 1 ergeben. Wir interpretieren z. B. Prognose=[0.10, 0.85, 0.05] als 10% Wahrscheinlichkeit, dass Klasse 1 vorliegt, 85% Wahrscheinlichkeit, dass Klasse vorliegt und 5% Wahrscheinlichkeit, dass Klasse 3 vorliegt

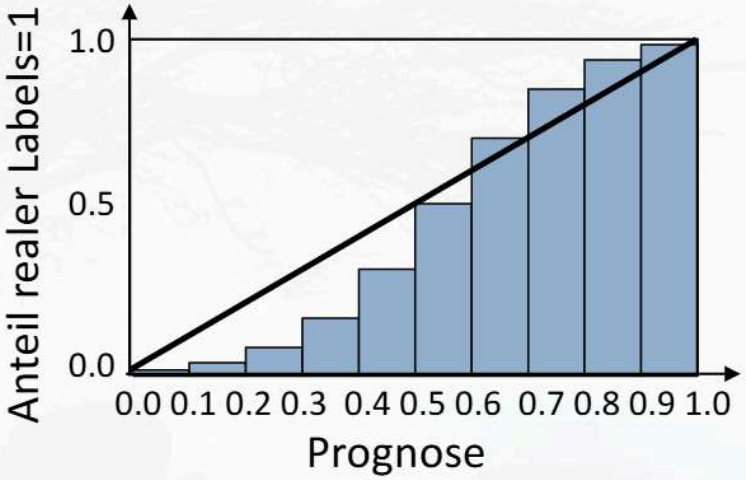
Bei der Binärklassifikation gibt es nur ein Label zwischen 0 und 1. Wir interpretieren z. B. 0.8 als 80% Wahrscheinlichkeit, dass Label=1 vorliegt und 20% Wahrscheinlichkeit, dass Label=0 vorliegt

Um sicherzustellen, dass das Ergebnis tatsächlich so zu interpretieren ist, kann man vergleichen wie viele Datenpunkte mit gleicher Prognose tatsächlich Label=1 bzw. Label=0 haben

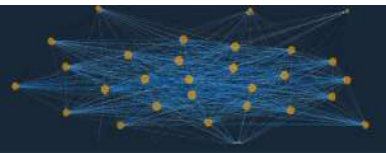
10 verschiedene Datenpunkte sollen ungefähr mit 0.8 prognostiziert sein, dann erwarten wir, dass 8 dieser Datenpunkte das reale Label=1 und 2 Datenpunkte das reale Label=0 haben

Calibration_curve aus scikit-learn vergleicht die Prognose mit dem Anteil der wahren Labels in verschiedenen Bereichen (der Bins) der Prognose

Durch ein Anpassungsmodell kann die Prognose kalibriert werden

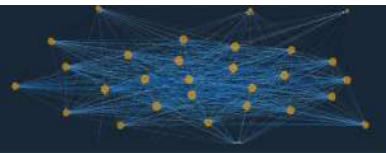


Übung Digits-Datensatz



- Lade den Digits-Datensatz von scikit-learn (sklearn.datasets.load_digits.html)
- # Stelle die ersten 10 Datenpunkte grafisch als Bild dar
- # Erstelle 64 Scatterplots für den Zusammenhang zwischen Label und den einzelnen Merkmalen
- Skaliere den Wertebereich der Merkmale auf Null bis Eins
- # Zähle wie viele Klassen und wie viele Datenpunkte pro Klasse es gibt
- Führe ein Onhotencoding der Labels durch
- Teile den Datensatz in Trainings- und Validierungsdaten auf
- Erstelle ein Sequential-Modell mit Keras mit einer Input-Schicht und weiteren Dense-Schichten
- Wähle für die Outputschicht die richtige Anzahl an Neuronen
- Gib dem Modell durch den Compile-Befehl die Loss-Funktion und den Optimizer an
- Fitte das Modell auf den Trainingsdaten und beurteile es mit den Validierungsdaten
- # Visualisiere die Lernkurven die als History nach dem Fit vom Modell abgerufen werden können
- Beurteile ob Overfitting oder Underfitting vorliegt
- # Optimiere die Hyperparameter des Modells
- # Erzeuge für die ersten 10 Validierungsdatenpunkte eine Prognose und gebe sie mit Wahrscheinlichkeiten aus
- # Erzeuge für die Validierungsdaten eine Confusion-Matrix
- Experimente: Variiere die Aktivierungsfunktion der Ausgabeschicht ('relu', 'sigmoid', 'softmax'), welche ist am besten?
- Variiere die Loss-Funktion ('mse', 'categorical_crossentropy') und vergleiche die Lernkurven

Übungsvorschläge



- Eigene neuronale Netze mit Keras anlegen z.B. für Wein-Datensatz https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_wine.html
- Sequential, Functional und Subclassing-Methode nachvollziehen und evaluieren (Kursbuch Kapitel 10, https://towardsdatascience.com/3-ways-to-create-a-machine-learning-model-with-keras-and-tensorflow-2-0-de09323af4d3)
- Mit Layer-Eigenschaften vertraut machen https://keras.io/api/layers/core_layers/dense/
- Wichtigste Unterschiede bei Regression und Klassifikation, Aktivierungs- und Loss-Funktionen nachvollziehen: Eine übersichtliche Tabelle (Spickzettel) zusammenstellen
- Eine lineare Regression mit Keras durchführen (neuronales Netz mit einer Schicht, einem einzelnen Neuron)
- Logistische Regression mit Keras (neuronales Netz mit einem einzelnen Neuron, Aktivierungsfunktion = sigmoid) Z.B. https://scikit-learn.org/stable/auto_examples/linear_model/plot_logistic.html#sphx-glr-auto-examples-linear-model-plot-logistic-py