Studienprojekt Data Science: Neuronale Netze zur Klassifikation von kardiovaskulären Ereignissen

Marc Karic

5. März 2024

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung und Ziel des Projektes

Im Rahmen eines verpflichtenden Studienprojektes im Bachelorstudiengang Data Science, habe ich zu dem Thema Neuronale Netze recherchiert und untersucht, wie Neuronale Netze für die Analyse von Patientendaten am Beispiel einer speziellen Fragestellung für konkrete Daten aus der UK Biobank genutzt werden können. Das Ziel dieses Reports soll sein, die Ergebnisse und Erkenntnisse aus dem Projekt darzustellen.
Um die Arbeitsweise eines Neuronalen Netzes zu verstehen, ist es natürlich wichtig, sich mit zentralen Komponenten von Neuronalen Netzen, wie unter anderem mit Forward- & Backpropagation, Aktivierungsfunktionen und Regularisierung zu beschäftigen. Zusätzlich habe ich zur Vertiefung zu Convolutional Neural Networks (CNN) recherchiert. Außerdem habe ich ein bekanntes Einsteigerbeispiel (MNIST Datensatz) vorbereitet, um die Anwendung eines CNN mit *Tensorflow* und *Keras* in Python zu demonstrieren. An den konkreten Daten der UK Biobank wurde versucht, ein Neuronales Netz zu bauen und zu trainieren, welches anhand ausgewählter Patientendaten voraussagen kann, ob ein Patient innerhalb der nächsten 10 Jahre ein kardiovaskuläres Ereignis (Hirn- und/oder Herzinfarkt) haben wird. Hier war das Ziel herauszufinden, wie gut die Vorhersagen des Netzes sind. Anhand dieses Ergebnisses kann die Qualität des Netzes mit anderen Tools (z.B. dem HeartScore) verglichen werden. Um dies zu visualisieren, kamen Methoden der ROC-Analyse zum Einsatz.

2 Das künstliche Neuron

Ein künstliches Neuronales Netz besteht unter anderem aus künstlichen Neuronen. Es bekommt einen Inputvektor der Dimension d $(x_1, \ldots, x_d)$ übergeben, welcher die Eigenschaften (*Features*) eines einzelnen Individuums enthält. In unserem Beispiel wären das alle medizinischen Werte eines Patienten, die wir dem Netz übergeben wollen. Ein Invidiuum mit dessen *Features* nennt man auch Datenpunkt. Darauf folgen d gerichtete Kanten, welche die jeweiligen Gewichte $(w_1, \ldots, w_d)$ enthalten. Die Startwerte dieser Gewichte werden am Anfang zufällig gewählt. Es existiert eine Transferfunktion g, auch Aktivierungsfunktion genannt, welche als Eingabe die Linearkombination aus Inputvektor, Gewichten und einem Skalar b, auch *Bias* genannt, erhält. Diese Transferfunktion entscheidet nun, ob $\sum_i^d w_i \cdot x_i + b > 0$ ist und somit ein y=1 liefert. Wenn dies nicht der Fall ist, wird stattdessen ein y=0 ausgegeben [4, S.288-289]. Kurz also:

$$g\left(\sum_i^d x_i \cdot w_i + b\right) = 1_{\{\sum_i^d x_i \cdot w_i + b > 0\}}$$

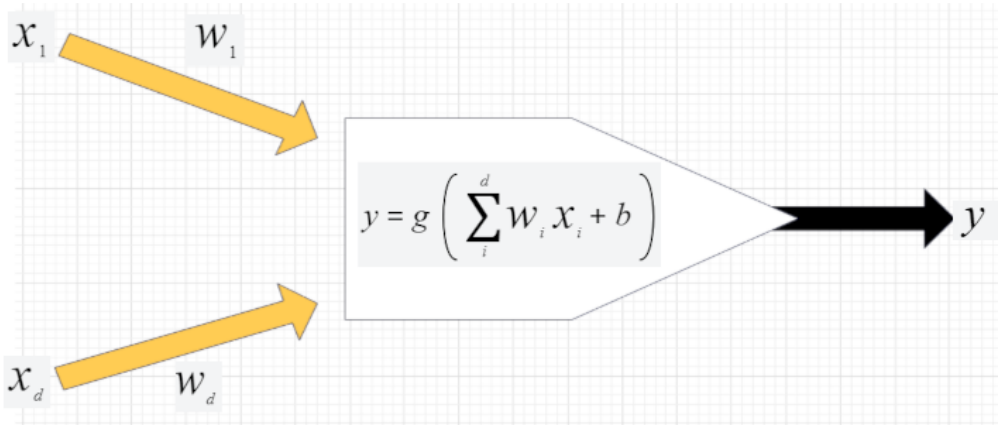

Abbildung 1: Das künstliche Neuron
Quelle: Ertel (2021:289); leicht verändert

3 Topologie eines Neural Networks

Um zu verstehen, wie ein Neuronales Netz funktioniert, ist es zunächst hilfreich zu verstehen, wie ein solches Netz aufgebaut ist. Eine vereinfachte Darstellung ist in Abbildung 2 zu finden. Ein Neuronales Netz hat L Schichten (auch Layer genannt, hier 5 Schichten), wobei jede Schicht meist mehrere Neuronen besitzt (ein Neuron ist jeweils mit gelb markiert), in denen jeweils (eventuell verschiedene) Aktivierungsfunktionen (hier die Knoten, die mit f betitelt sind) angewandt werden, die das Skalarprodukt z^1, z^2 und z^3 auswerten. Es erwartet immer einen x_d großen Inputvektor, der die jeweiligen Features eines einzelnen Datenpunktes enthält. Jeder einzelne Input hat dann gerichtete Kanten, auch Gewichte (W^1, W^2 und W^3) genannt, die mit einem Knoten verbunden sind. Dieser wertet dann die erhaltene Information aus und gibt das Ergebnis (a^1, a^2 und a^3) weiter. Jeder Input ist mit jedem Knoten über eine Kante verbunden. Wenn jeder Knoten mit jedem Folgeknoten verbunden ist, erhält ein solches Netz den Namen *Fully Connected Network*. Die Layer zwischen Input- und Outputlayern nennt man außerdem *Hidden Layer*. Schlussendlich gibt es einen Outputvektor $(y_1, \ldots, y_d)$, welcher das Ergebnis der vorherigen Skalarprodukte und Anwendungen der jeweiligen Aktivierungsfunktionen enthält. Dieser Output ist eine Voraussage eines bestimmten *Targets*, welches zuvor gewählt wird und auf welches das Netz trainiert wird. In unserem konkreten Beispiel wäre das *Target* eine Ausgabe zwischen 0 und 1, welche aussagt, ob ein Patient ein kardiovaskuläres Ereignis in den nächsten 10 Jahren haben wird. Die Schichten können unterschiedlich viele Neuronen haben, wobei mehr Neuronen und mehr Schichten auch eine höhere Komplexität des Netzes bedeutet. Dies kann zu einer längeren Rechenzeit führen und außerdem könnte sich das Netz an die Trainingsdaten, welche das Netz erhält um zu lernen, überanpassen (*overfitten*) und somit fremde Daten häufiger falsch einordnen. [3, S.9-10]

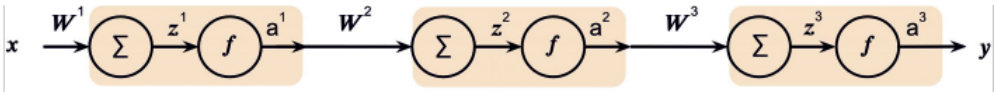

Abbildung 2: Vereinfachte Darstellung eines Neural Networks
Quelle: Drori (2023:10)

4 Forwardpropagation

Die Forwardpropagation beschreibt letztendlich das "Durchfüttern"der Daten durch das Neuronale Netz und würde für das Beispiel in Abbildung 3 folgendermaßen funktionieren:
Hier würde der obere Knoten (z_1^1) das Skalarprodukt aus den Inputs und den Gewichten $x_1 \cdot w_{11}^1 + x_2 \cdot w_{21}^1 + x_3 \cdot w_{31}^1$ erhalten. Dabei gibt das i von w_{ij}^k an, von welchem Neuron die Kante ausgeht, das j, zu welchem Neuron die Kante verläuft und das k das k-te Layer an. Die Aktivierungsfunktion f wertet diese Ausgabe dann aus, wodurch a_1^1 entsteht. Nun wird erneut ein Skalarprodukt mit a_1^1 und a_2^1 und den jeweils darauffolgenden Gewichten $W_{11}^2, W_{12}^2, W_{21}^2$ und W_{22}^2 zu z_1^2 und z_2^2 berechnet und an eine Aktivierungsfunktion f übergeben, wodurch dann wiederum a_1^2 und a_2^2 entstehen. Diese werden nun letztendlich mit den jeweiligen Gewichten W_1^3 und W_2^3 multipliziert und addiert, wodurch die Ausgabe y entsteht [3, S.11-14].

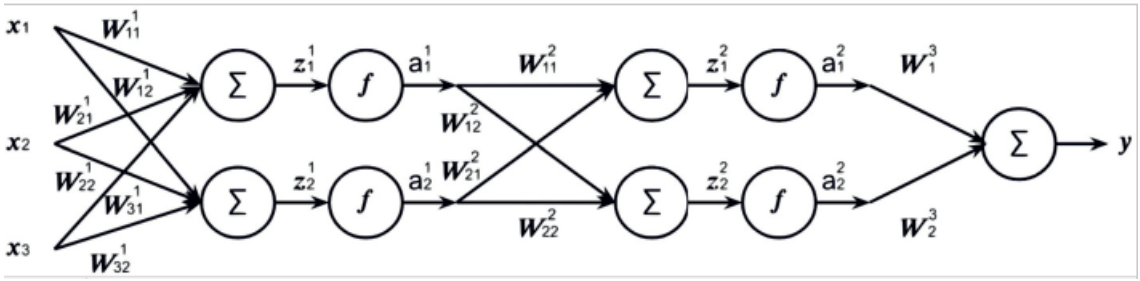

Abbildung 3: Beispiel eines Fully Connected Neural Networks
Quelle: Drori (2023:12)

Für das gegebene Beispiel in Abbildung 3 kann eine Formel für die Forwardpropagation formuliert werden, wobei F das gesamte Neuronale Netz als Funktion von $\mathbb{R}^3$ nach $\mathbb{R}^3$ repräsentiert, die für einen Inputvektor $x_i \in \mathbb{R}^3$ den zugehörigen Output y ergibt. F wird durch die gezeigten Komponenten des Neuronalen Netzes beginnend mit Layer 3 sukzessive bis zu Layer 0 (Inputvektor) ersetzt und wird damit schließlich in stark verschachtelte Hintereinanderausführung mehrerer Skalarprodukte und nicht-linearen Transformationen (f) dargestellt.

$$
\begin{aligned}
y &= F(x_{i1}, x_{i2}, x_{i3}) \\
&= a_{i1}^2 \cdot w_1^3 + a_{i2}^2 \cdot w_2^3 \\
&= f(z_1^2) \cdot w_1^3 + f(z_2^2) \cdot w_2^3 \\
&= f\{a_{i1}^1 \cdot w_{11}^2 + a_{i2}^1 \cdot w_{21}^2\} \cdot w_1^3 \\
&+ f\{a_{i1}^1 \cdot w_{12}^2 + a_{i2}^1 \cdot w_{22}^2\} \cdot w_2^3 \\
&= \{f[f(x_{i1} \cdot w_{11}^1 + x_{i2} \cdot w_{21}^1 + x_{i3} \cdot w_{31}^1) \cdot w_{11}^2 + f(x_{i1} \cdot w_{12}^1 + x_{i2} \cdot w_{22}^1 + x_{i3} \cdot w_{32}^1) \cdot w_{21}^2]\} \cdot w_1^3 \\
&+ \{f[f(x_{i1} \cdot w_{11}^1 + x_{i2} \cdot w_{21}^1 + x_{i3} \cdot w_{31}^1) \cdot w_{12}^2 + f(x_{i1} \cdot w_{12}^1 + x_{i2} \cdot w_{22}^1 + x_{i3} \cdot w_{32}^1) \cdot w_{22}^2]\} \cdot w_2^3
\end{aligned}
$$

5 Backpropagation

Die Backpropagation ist der Prozess des Lernens des Neuronalen Netzes. Da es auf der Grundlage der korrekten Trainingsdaten lernt, gehören Neuronale Netze zum sogenannten *supervised Learning*. Daher ist es notwendig, ein *Target* für die Daten zu haben, auf dem diese trainiert werden. Das bedeutet, dass das Netz einen Vergleich zwischen seinem Output und dem Target-Wert herstellen muss, um zu bewerten, wie stark die beiden voneinander abweichen. Dieser Vergleich erfolgt über eine sogenannte *Lossfunktion*, die ein Abweichungsmaß zwischen Target und Output berechnet. Eine solche Lossfunktion könnte beispielsweise der mittlere quadratische Fehler sein:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i^{target} - y_i^{predicted})^2$$

Anhand dieses Fehlers werden dann die Gewichte angepasst, wobei y_i^{target} der jeweilige Target-Wert und $y_i^{predicted}$ die jeweilige Vorhersage des Neuronalen Netzes ist, um im Optimalfall bei der nächsten *Epoche* des Trainierens mit der Ausgabe im Vergleich zum *Target* richtigzuliegen (also keinen Fehler zu haben oder den Fehler möglichst minimal zu halten). Eine *Epoche* beschreibt eine Forward- und Backpropagation mit allen Daten. Die Backpropagation löst also ein Minimierungsproblem. Der Ablauf sieht nach der Forwardpropagation und Berechnung des Fehlers wie folgt aus: [2, S.138-141] [3, S.21-33]
Für jeden Layer l = L, ... , 1 (wobei L der letzte Layer ist):

- Nutze die Kettenregel, um den Gradienten des Outputs in Bezug auf den Input zu berechnen.
- Aktualisiere die Gewichte in dem aktuellen Layer mit folgender Regel:
- $w^l = w^l - \alpha \frac{\partial \mathcal{L}}{\partial w^l}$
  Wobei w^l die Gewichte, l das *l-te* Layer, α die Lernrate und $\mathcal{L}$ der Loss bezüglich der Gewichte sind. Die Lernrate ist die Schrittweite, mit der die Veränderung für alle Gewichte erfolgen soll und kann durch verschiedene Methoden entweder während des Lernens verändert werden oder etwa mittels Kreuzvalidierung möglichst optimal bestimmt werden.

Wie sich die Gradienten analytisch berechnen lassen, sehen wir nun für jeweils ein Gewicht in jedem Layer. Die Gradienten beziehen sich auf das Netz aus Abbildung 3:

Dritter Layer:

$$\begin{aligned}
\frac{\partial}{\partial w_1^3}\frac{1}{n}\sum_{i=1}^{n}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3}))^2 &= \frac{1}{n}\sum_{i=1}^{n}\frac{\partial}{\partial w_1^3}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3}))^2 \\
&= -\frac{1}{n}\sum_{i=1}^{n} 2 \cdot (y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot \frac{\partial}{\partial w_1^3} \cdot (a_{i1}^2 \cdot w_1^3 + a_{i2}^2 \cdot w_2^3) \\
&= -\frac{1}{n}\sum_{i=1}^{n} 2 \cdot (y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot a_{i1}^2
\end{aligned}$$

Zweiter Layer:

$$\begin{aligned}
\frac{\partial}{\partial w_{11}^2}\frac{1}{n}\sum_{i=1}^{n}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3}))^2 &= -\frac{1}{n}\sum_{i=1}^{n} 2 \cdot (y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot \frac{\partial}{\partial w_{11}^2} \cdot (a_{i1}^2 \cdot w_1^3 + a_{i2}^2 \cdot w_2^3) \\
&= -\frac{1}{n}\sum_{i=1}^{n} 2 \cdot (y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot \frac{\partial}{\partial w_{11}^2}(f(z_{i1}^2) \cdot w_1^3 + f(z_{i2}^2) \cdot w_2^3) \\
&= -\frac{1}{n}\sum_{i=1}^{n} 2 \cdot (y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot \frac{\partial}{\partial w_{11}^2}(f(a_{i1}^1 \cdot w_{11}^2 + a_{i2}^1 \cdot w_{21}^2) \cdot w_1^3 \\
&+ f(a_{i1}^1 \cdot w_{12}^2 + a_{i2}^1 \cdot w_{22}^2) \cdot w_2^3) \\
&= -\frac{1}{n}\sum_{i=1}^{n} 2 \cdot (y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \\
&\cdot (f'(a_{i1}^1 \cdot w_{11}^2 + a_{i2}^1 \cdot w_{21}^2) \cdot w_1^3 \cdot a_{i1}^1)
\end{aligned}$$

Erster Layer:

$$
\begin{aligned}
\frac{\partial}{\partial w_{11}^1}\frac{1}{n}\sum_{i=1}^{n}(y_i^{target}-F(x_{i1},x_{i2},x_{i3}))^2 &= -\frac{1}{n}\sum_{i=1}^{n}2\cdot(y_i^{target}-F(x_{i1},x_{i2},x_{i3}))\cdot\frac{\partial}{\partial w_{11}^1}(a_{i1}^2\cdot w_1^3+a_{i2}^2\cdot w_2^3)\\
&= -\frac{1}{n}\sum_{i=1}^{n}2\cdot(y_i^{target}-F(x_{i_1},x_{i2},x_{i3}))\cdot\frac{\partial}{\partial w_{11}^1}(f(z_{i1}^2)\cdot w_1^3+f(z_{i2}^2)\cdot w_2^3)\\
&= -\frac{1}{n}\sum_{i=1}^{n}2\cdot(y_i^{target}-F(x_{i1},x_{i2},x_{i3}))\cdot\frac{\partial}{\partial w_{11}^1}(f(a_{i1}^1\cdot w_{11}^2+a_{i2}^1\cdot w_{21}^2)\cdot w_1^3\\
&+ f(a_{i1}^1\cdot w_{12}^2+a_{i2}^1\cdot w_{22}^2)\cdot w_2^3)\\
&= -\frac{1}{n}\sum_{i=1}^{n}2\cdot(y_i^{target}-F(x_{i1},x_{i2},x_{i3}))\\
&\cdot\frac{\partial}{\partial w_{11}^1}(\{f[f(x_{i1}\cdot w_{11}^1+x_{i2}\cdot w_{21}^1+x_{i3}\cdot w_{31}^1)\cdot w_{11}^2\\
&+ f(x_{i1}\cdot w_{12}^1+x_{i2}\cdot w_{22}^1+x_{i3}\cdot w_{32}^1)\cdot w_{21}^2]\}\cdot w_1^3\\
&+ \{f[f(x_{i1}\cdot w_{11}^1+x_{i2}\cdot w_{21}^1+x_{i3}\cdot w_{31}^1)\cdot w_{12}^2\\
&+ f(x_{i1}\cdot w_{12}^1+x_{i2}\cdot w_{22}^1+x_{i3}\cdot w_{32}^2)\cdot w_{22}^2]\}\cdot w_2^3)\\
&= -\frac{1}{n}\sum_{i=1}^{n}2\cdot(y_i^{target}-F(x_{i1},x_{i2},x_{i3}))\\
&\cdot f'\{f(x_{i1}\cdot w_{11}^1+x_{i2}\cdot w_{21}^1+x_{i3}\cdot w_{31}^1)\cdot w_{11}^2\\
&+ f(x_{i1}\cdot w_{12}^1+x_{i2}\cdot w_{22}^1+x_{i3}\cdot w_{32}^1)\cdot w_{21}^1\}\cdot w_1^3\\
&\cdot[f'(x_{i1}\cdot w_{11}^1+x_{i2}\cdot w_{21}^1+x_{i3}\cdot w_{31}^1)\cdot w_{11}^2\\
&+ f'(x_{i1}\cdot w_{12}^1+x_{i2}\cdot w_{22}^1+x_{i3}\cdot w_{32}^1)\cdot w_{21}^1]\\
&\cdot(x_{i1})\\
&+ f'\{f(x_{i1}\cdot w_{11}^1+x_{i2}\cdot w_{21}^1+x_{i3}\cdot w_{31}^1)\cdot w_{12}^2\\
&+ f(x_{i1}\cdot w_{12}^1+x_{i2}\cdot w_{22}^1+x_{i3}\cdot w_{32}^1)\cdot w_{22}^2\}\cdot w_2^3\\
&\cdot[f'(x_{i1}\cdot w_{11}^1+x_{i2}\cdot w_{21}^1+x_{i3}\cdot w_{31}^1)\cdot w_{12}^2\\
&+ f'(x_{i1}\cdot w_{12}^1+x_{i2}\cdot w_{22}^1+x_{i3}\cdot w_{32}^1)\cdot w_{22}^2]\\
&\cdot(x_{i1})
\end{aligned}
$$

6 Aktivierungsfunktionen

Für Neuronale Netze gibt es einige Aktivierungsfunktionen. Sie sind mathematische Operationen, die in jedem Neuron eines neuronalen Netzes angewendet werden. Sie sind dafür verantwortlich, welche Ausgabe ein Neuron erzeugt, da sie das Skalarprodukt der Eingabe und der Gewichte erhalten. Aktivierungsfunktionen sind in der Regel non-linear. Würde man ausschließlich lineare Aktivierungsfunktionen in einem Netz mit mehreren Layern verwenden, könnte man das Netz auf einen Layer reduzieren, da die Komplexität die Gleiche wäre. Im Vergleich zu einem Netz mit non-linearen Aktivierungsfunktionen würde somit das Potenzial eines komplexen Netzes verloren gehen. Im Folgenden

werden einige Aktivierungsfunktionen vorgestellt und welchen Nutzen sie haben. [3, S.16-19]

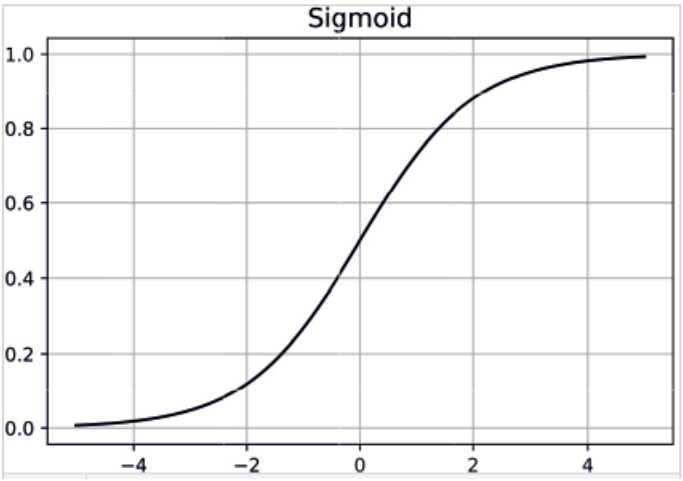

Abbildung 4: Sigmoid
Quelle: Drori (2023:16)

Die Sigmoid Funktion skaliert den erhaltenden Input auf einen Wert zwischen 0 und 1.

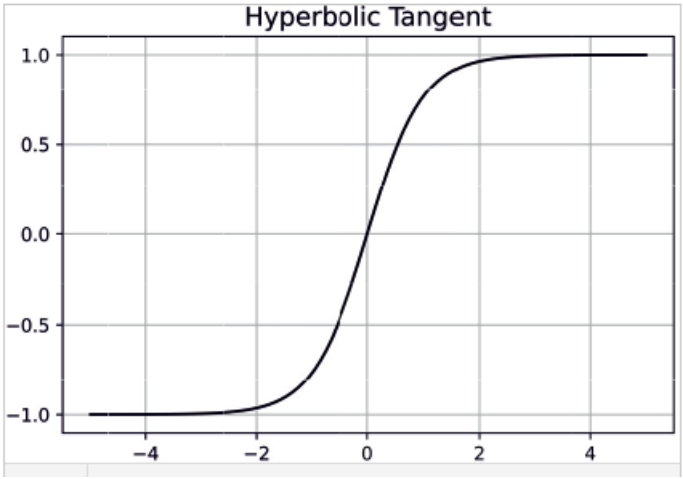

Abbildung 5: Hyperbolic Tangent (tanh)
Quelle: Drori (2023:17)

Da tanh Werte zwischen -1 und 1 liefert, wird sie häufig in Hidden Layers verwendet, um die Daten zu zentrieren.

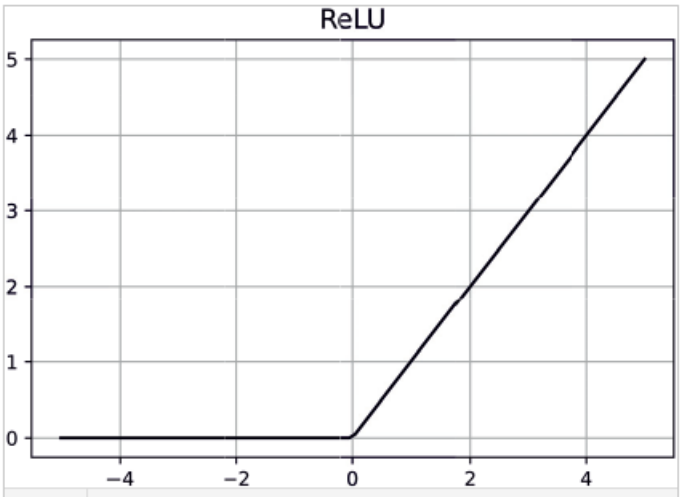

Abbildung 6: Rectified Linear Unit (ReLU)
Quelle: Drori (2023:17)

ReLU hat den großen Vorteil, leicht berechnet zu werden, da sie weniger mathematische Operationen verwendet. ReLU wird zu dem am häufigsten als Aktivierungsfunktion verwendet, da sie sich in vergangenen Netzen gut bewiesen hat.

Softmax (hier nicht gezeigt) wird hingegen für Multiclass Klassifizierung im Ausgabelayer verwendet. Sie gibt eine Wahrscheinlichkeit für jede Klasse aus.

7 Regularisierung

Regularisierung ist eine Möglichkeit, Overfitting (d.h. eine Überanpassung an die Trainingsdaten) zu verhindern und diese zu bestrafen. Dazu gibt es mehrere Möglichkeiten, um dies zu bewältigen. [3, S.58-65]

Weight Decay
Beim Weight Decay wird ein zusätzlicher Term an die Lossfunktion angehängt, welche häufig die L2-Norm ist.

$$||w||^2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$$

Der Weight Decay soll große Gewichte und große Gradienten verhindern. Diese könnten dafür sorgen, dass das Netz nicht konvergieren kann und somit nicht die optimalen Gewichte finden kann. $\mathcal{L}$ ist hier der Loss und λ ist ein Skalar zwischen 0 und 1, was im Endeffekt die Stärke der Bestrafung festlegt.

$$\mathcal{L} = \mathcal{L} + \lambda \cdot ||w||^2$$

Dropout
Dropout sorgt dafür, dass während des Trainings zu einer bestimmten Wahrscheinlichkeit Neuronen auf null gesetzt werden. Dies hilft der Robustheit gegenüber *Overfitting*.

Data Augmentation
Data Augmentation sorgt dafür, mehr Trainingsdaten durch Transformation zu erschaffen, was das Neuronale Netz robuster macht. Die Transformation sieht je nach Anwendung unterschiedlich aus, allerdings kann in unserem konkreten Beispiel etwas Rauschen auf die Daten angewandt werden. In Anwendungen der Bildklassifikation ist es z.B. üblich die Bilder etwas zu rotieren.

Batch Normalization
Batch Normalization ist eine weitere Methode, um große Gradienten zu verhindern. Es skaliert die Daten mit dem Faktor $\frac{1}{\sqrt{n}}$,wobei n die Größe eines Batches ist. Für das Training neuronaler Netze kann die *Batchsize* festgelegt werden. Diese bestimmt, wie viele Trainingsdaten dem Netz übergeben werden. Somit würde die Gewichtsanpassung nach jedem Batch geschehen.

8 Struktur eines Convolutional Neural Networks

Convolutional Neural Networks (kurz CNN) werden häufig z.B. für die Bildklassifikation verwendet. Es besitzt Convolutional und Pooling Layer, welche in gewissermaßen zur Vorverarbeitung der Daten dienen, bevor sie in ein Fully Connected Network weiterverarbeitet werden, wie wir es bereits kennen. Dabei wird zunächst ein Convolutional Layer und danach ein Pooling Layer benutzt. Allerdings kann dies auch mehrfach wiederholt werden, um genauere *Features* eines Bildes hervorzuheben.

8.1 Convolution

Die Convolution ist im Endeffekt nichts Anderes, als eine mathematische Operation, welche zwei Container mit Information miteinander vereinen kann. Dies geschieht über Matrix-Matrix-Multiplikation. In einem CNN wird auf die Inputdaten ein Filter angewandt, welcher bestimmte Eigenschaften hervorheben soll. Man nennt das Ergebnis daraus auch *Feature map*. [3, S.70-78]

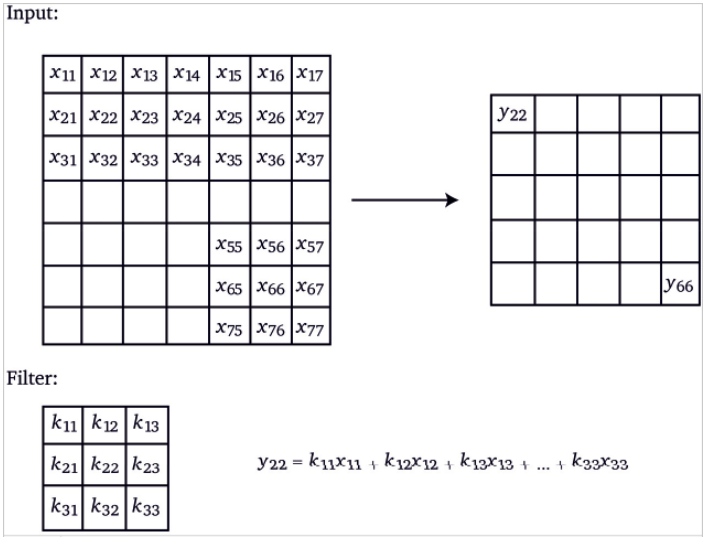

Abbildung 7: 2-dimensionelle Convolution
Quelle: Drori (2023:74)

8.2 Padding und Striding

Padding und Striding sind hilfreiche Techniken, die häufig in CNNs benutzt werden. Padding erweitert am Rand der Matrix künstliche Matrixeinträge, um die Information, die am Rande der Matrix enthalten ist, während der *Convolution* nicht zu verlieren. Striding hingegen beschreibt die Schrittweise, mit der der Filter über die Matrix läuft. Je nachdem, wie groß der Stride gewählt wird, wird sich auch die Größe der Outputmatrix verändern. [3, S.70-75]

8.3 Pooling

Pooling wird nach einem Convolutional Layer verwendet, um die Dimension der *Feature map* zu reduzieren und somit die Anzahl der Trainingsparameter zu verkleinern. [3, S.78-80]

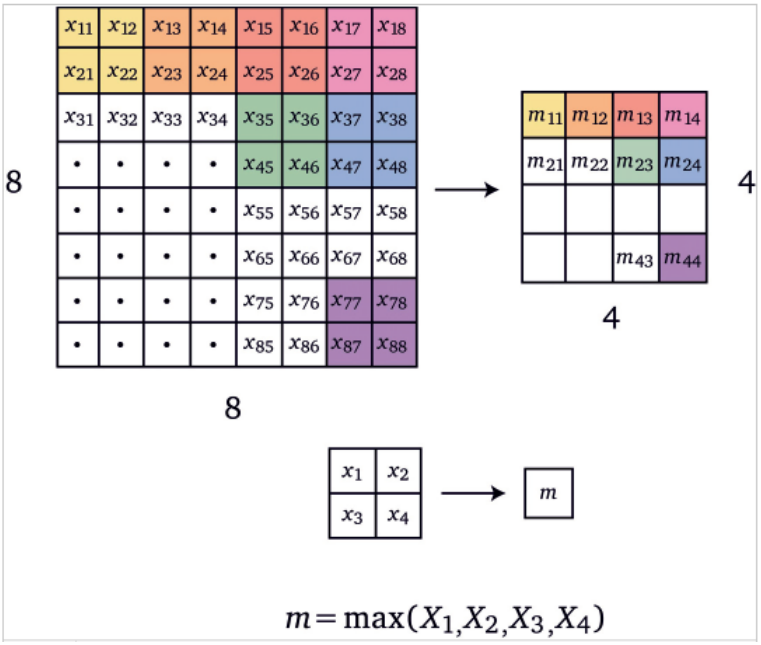

Abbildung 8: Pooling
Quelle: Drori (2023:79)

9 Einsteigerbeispiel eines CNNs in Python

Im Folgenden werden wir uns mit einem klassischen Einsteigerbeispiel für Neuronale Netze beschäftigen. Die MNIST Datenbank ist eine öffentliche Datenbank, mit 70.000 handgeschriebenen Ziffern im 28x28 Pixel Format, wobei jeder Pixel jeweils eine Graustufe von 0-255 besitzt. Die Daten sind bereits vorverarbeitet, es ist also keine manuelle Bereinigung, Kodierung oder Ähnliches notwendig.

9.1 Import aller notwendigen Libraries

Zuerst müssen alle wichtigen Libraries importiert werden, die für dieses Beispiel notwendig sind. Darunter zählt unter Anderem *Tensorflow* und *Keras*, welche helfen, das Neuronale Netz aufzubauen. Außerdem hilft *matplotlib* die Daten zu visualisieren.

```
import tensorflow as tf
from tensorflow import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Dropout
from keras.layers import Activation, Flatten
from matplotlib import pyplot as plt
%matplotlib inline
from IPython.display import clear_output
import numpy as np
import random
```

9.2 Veranschaulichung einiger handschriftlichen Zahlen aus dem MNIST Dataset

Hier werden zunächst nur die Daten geladen und einige handschriftliche Zahlen geplottet, um zu zeigen, wie die Daten überhaupt aussehen. Außerdem wird die Dimensionen der Trainings- und Testdaten ausgegeben.

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)

# Plot einiger handgeschriebenen Ziffern
f, axes = plt.subplots(1, 10, sharey=True,figsize=(10,10))
for i,ax in enumerate(axes.flat):
ax.axis('off')
ax.imshow(X_train[i,:,:,0],cmap="gray")
```

Abbildung 9: Einige handschriftliche Zahlen aus dem MNIST Dataset
Quelle: Eigene Darstellung, Datengrundlage: MNIST Datenbank (2023)

```
# Dimensionen der Trainings- und Testdaten.
# X ist hier jeweils der Input und y das Target.
print("X_train Shape: ", X_train.shape)
print("y_train Shape: ", y_train.shape)
print("X_test Shape: ", X_test.shape)
print("y_test Shape: ", y_test.shape)
```

```
X_train Shape:  (60000, 28, 28, 1)
y_train Shape:  (60000,)
X_test Shape:  (10000, 28, 28)
y_test Shape:  (10000,)
```

9.3 Erstellung des Neuronalen Netzes

Nun kann das Netz auch schon erstellt werden. Hierfür wird einfach das *Sequential* Model von *Tensorflow* geladen und dann wird das Netz Layer für Layer mittels *model.add()* aufgebaut. Schlussendlich ruft man *model.compile()* auf, um das Netz zusammenbauen zu lassen, wo man die passende *Lossfunktion*, *Optimizer* (also der Algorithmus, der für die Gewichtsanpassung zuständig ist) und *Metrik* auswählen kann.

```
def create_base_model(num_classes = 10):
'''
create_base_model erstellt das Neuronale Netz.
Hierfuer kann einfach ein Model geladen werden und nacheinander
die gewuenschten Layer mit ihren Argumenten hinzugefuegt werden.
```

```
    Flatten rollt sozusagen das mehrdimensionale Array
    auf in ein einzelnes Array, um es dann in ein Dense Layer uebergeben
    zu koennen.
    Compile baut letztendlich das Model zusammen.
    '''
    model = Sequential()

    input_shape = (28, 28, 1)

    model.add(Conv2D(32, kernel_size = (5,5), strides = (2,2),
                     activation = tf.nn.relu, input_shape = input_shape))
    model.add(MaxPooling2D(pool_size = (2,2), strides = (2,2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation = tf.nn.relu))
    model.add(Dense(num_classes, activation = tf.nn.softmax))

    model.compile(loss = "MeanSquaredError",
                  optimizer  ="RMSprop",
                  metrics = ["accuracy"])
    return model
```

Der folgende Codeausschnitt wird benötigt, um den Loss und die Accuracy des Modells während des Lernens plotten zu können. Es ist also nur für Demonstrationszwecke notwendig und daher auch nicht wichtig zu verstehen.

```
class PlotLearning(keras.callbacks.Callback):
def on_train_begin(self, logs={}):
self.metrics = {}
for metric in logs:
self.metrics[metric] = []


def on_epoch_end(self, epoch, logs={}):
for metric in logs:
if metric in self.metrics:
self.metrics[metric].append(logs.get(metric))
else:
self.metrics[metric] = [logs.get(metric)]

# Plotting
metrics = [x for x in logs if 'val' not in x]

f, axs = plt.subplots(1, len(metrics), figsize=(15,5))
clear_output(wait=True)

for i, metric in enumerate(metrics):
axs[i].plot(range(1, epoch + 2),
self.metrics[metric],
```

```
label=metric, marker='o')
if logs['val_' + metric]:
axs[i].plot(range(1, epoch + 2),
self.metrics['val_' + metric],
label='val_' + metric, marker='o')

axs[i].legend()
axs[i].grid()

axs[0].set_title("Loss")
axs[1].set_title("Accuracy")
plt.tight_layout()
plt.show()
```

9.4 Ausführung des Modells

Hier beginnt nun das Training des Modells. Dafür wird das Modell aus **9.3** geladen und mit *model.fit()* wird das Training begonnen, wobei hier noch die Trainingsdaten, das Target, die Anzahl der *Epochen* und die *batch_size* übergeben werden.

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
# Normalisierung der Graustufenwerte auf den Wertebereich 0-1,
# um eine bessere Performance des Netzes zu erzielen.
X_train = X_train.astype("float32") / 255
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
X_test = X_test.astype("float32") / 255

y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

callbacks_list = [PlotLearning()]

model = create_base_model()
model.summary()
model.fit(X_train, y_train, validation_split = 0.2,
          epochs = 20, batch_size = 75, verbose = 2,
          callbacks=callbacks_list)
```

Mit *model.evaluate()* kann man die Performance des Netzes mit den Testdaten abfragen. In unserem Fall liegt der Loss bei 0.00159 und die Accuracy bei 0.99.

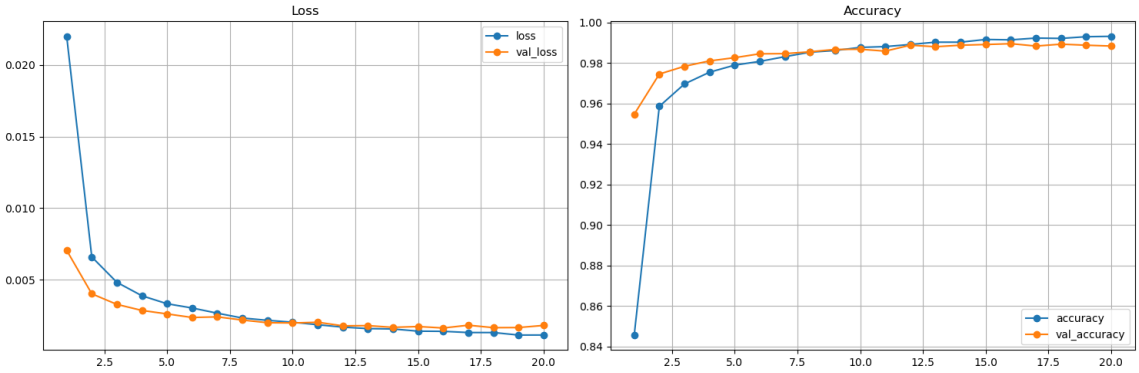

Abbildung 10: Trainings- und Validierungsloss und Accuracy während des Trainings
Quelle: Eigene Darstellung, Datengrundlage: MNIST Datenbank (2023)

9.5 Vergleich einiger zufälliger Bilder zur Ausgabe des Neuronalen Netzes

Der folgende Code zeigt, welche Zahlen das Netz für zufällige handschriftliche Zahlen voraussagt.

```
# Auch hier ist es nicht wichtig den Code zu verstehen,
# er dient nur zu Demonstrationszwecken.
figure = plt.figure(figsize=(10,2))
rand = random.randint(0, 50000)
print("Ausgabe des Neuronalen Netzwerkes: ")
for i in range(5):
figure.add_subplot(1,5,i+1)
plt.imshow(X_train[i+rand,:,:,0],cmap="gray")
plt.axis("off")
plt.suptitle("Zufaellige Bilder des MNIST Datasets")
print(np.squeeze(np.argmax(model.predict(X_train[i+rand].reshape(1,28,28,1),
      verbose = 0),axis=1),axis=0),end="\t")
```

Ausgabe des Neuronalen Netzwerkes:
5 6 7 7 9

Abbildung 11: Einige zufällige Bilder aus der MNIST Datenbank
Quelle: Eigene Darstellung, Datengrundlage: MNIST Datenbank (2023)

10 Anwendung eines Neuronalen Netzes zur Klassifikation von CVDs

10.1 Fragestellung und Datenstruktur der Patientendaten

In diesem Anwendungsbeispiel wurden die Daten der UK Biobank verwendet, welche Patientendaten von über 477.000 Probanden enthalten. Dafür wurden *Alter*, *Geschlecht*, *systolischer Blutdruck*, *Gesamt-Cholesterol*, *HDL-Cholesterol*, *LDL-Cholesterol*, *Raucher (Ja/Nein)*, *Datum der Baseline-Untersuchung*, *Datum für Herzinfarkt (falls vorhanden)* und *Datum für Hirninfarkt (falls vorhanden)* untersucht.
Da alle Patienten mindestens seit 10 Jahren in der Datenbank sind und diese Daten auch aktuell sind, ist es möglich, das *Ten Year Survival* der Patienten zu bestimmen, also ob ein Patient 10 Jahre, im Vergleich zu der Baseline-Untersuchung, einen Herz- und/oder Hirninfarkt erlitten hat. Dies ist also eine binäre Ausgabe: Klasse 0 hat innerhalb von 10 Jahren ein kardiovaskuläres Ereignis erlitten und Klasse 1 hat innerhalb von 10 Jahren kein kardiovaskuläres Ereignis erlitten.

10.2 Präprozessierung der Daten

Vorab mussten aber anhand der oben genannten *Features* einige Informationen abgeleitet werden. Unter anderem durfte nur das erste Datum eines Patienten benutzt werden, wenn dieser sowohl Herz- als auch Hirninfarkt hatte. Des Weiteren durften nur Patienten untersucht werden, deren Datum des kardiovaskulären Ereignisses nach der Baseline-Untersuchung liegt. Daraus ergibt sich schlussendlich eine Gesamtpopulation von 393.718 Patienten, mit 7 *Features* und einem *Target*.

	Age	Sex	SBP	Cholesterol	HDL	LDL	SmokingStatus	TenYearSurvival
0	65	F	187.0	6.515	1.546	4.115	0.0	1.0
1	64	M	138.0	6.633	1.239	4.336	1.0	0.0
2	45	F	105.0	5.034	2.292	2.655	0.0	1.0
3	57	M	126.0	5.678	1.125	3.912	1.0	1.0
4	66	F	115.0	5.095	2.296	2.359	1.0	1.0

Tabelle 1: Ausschnitt der Daten nach der Bereinigung
Quelle: Eigene Darstellung, Datengrundlage: UK Biobank (2023)

10.3 One-Hot Encoding und Normalisierung der Daten

Da ein Neuronales Netz keine kategorischen *Features* verarbeiten kann, ist es notwendig, diese zu kodieren. Eine einfache Umwandlung in Zahlen für jede Kategorie in dem jeweiligen *Feature* würde eine künstliche Reihenfolge innerhalb der Daten generieren, was sie im Endeffekt verfälschen würde. Deshalb gibt es unter anderem das sogenannte One-Hot Encoding, was das *Feature* in so viele *Features* aufteilt, wie es Kategorien innerhalb des *Features* gibt. So wird beispielsweise das *Feature Sex*, bei dem es die Kategorie *M* für männlich und *F* für weiblich gibt, in zwei Features aufgeteilt. In dem Falle *Sex_F* und *Sex_M*, welche durch das One-Hot Encoding binär gemacht wurden. Dadurch ist das Feature nun numerisch und nicht mehr kategorisch und kann dem Netz übergeben werden.

	0	1	2	3	4
Age	65.000	64.000	45.000	57.000	66.000
SBP	187.000	138.000	105.000	126.000	115.000
Cholesterol	6.515	6.633	5.034	5.678	5.095
HDL	1.546	1.239	2.292	1.125	2.296
LDL	4.115	4.336	2.655	3.912	2.359
TenYearSurvival	1.000	0.000	1.000	1.000	1.000
Sex_F	1.000	0.000	1.000	0.000	1.000
Sex_M	0.000	1.000	0.000	1.000	0.000
SmokingStatus_0.0	1.000	0.000	1.000	0.000	0.000
SmokingStatus_1.0	0.000	1.000	0.000	1.000	1.000

Tabelle 2: Ausschnitt der Daten nach dem One-Hot Encoding
Quelle: Eigene Darstellung, Datengrundlage: UK Biobank (2023)

Außerdem ist es hilfreich, die Daten zu normalisieren, da man somit eine bessere Konvergenz des Netzes erzielen kann und man zu dem vermeiden kann, dass die Gradienten immer größer werden und deshalb die Gewichtsänderung während des Trainings zu groß wird und das Netz somit nicht konvergieren kann. [3, S.33]

	0	1	2	3	4
Age	0.771429	0.742857	0.200000	0.542857	0.800000
SBP	0.606796	0.368932	0.208738	0.310680	0.257282
Cholesterol	0.356891	0.365375	0.250413	0.296714	0.254799
HDL	0.316168	0.242635	0.494850	0.215329	0.495808
LDL	0.403214	0.426426	0.249869	0.381893	0.218780
TenYearSurvival	1.000000	0.000000	1.000000	1.000000	1.000000
Sex_F	1.000000	0.000000	1.000000	0.000000	1.000000
Sex_M	0.000000	1.000000	0.000000	1.000000	0.000000
SmokingStatus_0.0	1.000000	0.000000	1.000000	0.000000	0.000000
SmokingStatus_1.0	0.000000	1.000000	0.000000	1.000000	1.000000

Tabelle 3: Ausschnitt der Daten nach dem One-Hot Encoding und Normalisierung
Quelle: Eigene Darstellung, Datengrundlage: UK Biobank (2023)

10.4 Aufteilung der Daten in Trainings-, Validierungs & Testdaten

Für das Erstellen eines Neuronalen Netzes ist es wichtig, die Daten in Trainings-, Validierungs- und Testdaten aufzuteilen. Die Trainingsdaten benötigt das Netz, um sich selbst während der Backpropagation zu verbessern und einen Vergleich zu dem *Target* zu haben. Die Validierungsdaten nutzt das Netz nicht zum Training und können daher gute Auskunft darüber geben, ob das Netz während des Trainings bereits *overfitten* wird, also es nicht dazu in der Lage ist zu generalisieren. Die Testdaten sind ebenfalls eine Überprüfungsmechanik des Netzes, um zu verstehen, ob das Netz in der Lage ist zu generalisieren. Diese werden nach dem Training dem Netz übergeben, um es zu evaluieren.
In der Regel machen die Trainingsdaten etwa 70-90% der gesamten Daten und die Validierungsdaten etwa 10-30% der Trainingsdaten aus.

10.5 Das Problem der Klassenverteilung

Nach der Präprozessierung wurde ein erstes Neuronales Netz trainiert. Während des Trainings hatte sich die Metriken nicht verändert, deshalb wurden zuerst die Hyperparameter verändert, um herauszufinden, ob das Netz eventuell in einem lokalen Minimum feststeckt. Nachdem das aber das

Problem nicht behob, wurde klar, dass es vermutlich mit der Klassenverteilung des *Targets* zusammenhängen muss. Die Klassenverteilung der präprozessierten Daten beträgt **96.4/3.7**. Aus diesem Grund wurden mehrere Klassenverteilungen ausgetestet, indem diese Verteilung künstlich verändert wurde, um dieses Problem genauer zu untersuchen:

- 50/50
- 60/40
- 70/30
- 80/20
- 90/10
- 95/5
- 96.4/3.7 (Verteilung der Basisdaten)

Das Ergebnis dieser Untersuchung wird in Kapitel **10.9.1** zusammengefasst.

10.6 Metriken

Für das Training des Netzes wurden einige Metriken verwendet. Diese werden in dem folgenden Abschnitt erläutert. Hierbei steht *TP* für die richtig positiven Werte, *FP* für die falsch positiven Werte, *TN* für die richtig negativen Werte und *FN* für die falsch negativen Werte. [1] [5]

		Predicted condition		
		Predictive Positive (PP)	Predicted Negative (PN)	
Actual condition	Positive (P)	True positive (TP)	False negative (FN)	Sensivity
	Negative (N)	False positive (FP)	True negative (TN)	Specificity
		Precision		

Precision
Precision wird häufig auch *positive predictive value* genannt und ist der Anteil der korrekt positiven Werten durch die korrekt positiven und negativen Werten.

$$Precision = \frac{TP}{TP + FP}$$

Recall und Sensitivity
Recall, auch *Sensitivity* genannt, ist der Anteil an korrekt positiven Werten durch die Anzahl der korrekt positiven Werten und falsch negativen Werten.

$$Recall/Sensitivity = \frac{TP}{TP + FN}$$

Specificity
Specificity, auch *true negative rate* genannt, ist der Anteil an korrekt negativen Werten durch die Anzahl der korrekt negativen Werte und falsch positiven Werten.

$$Specificity = \frac{TN}{TN + FP}$$

10.7 Hyperparametersuche

Da ein Neuronales Netz viele Stellschrauben (*Hyperparameter*) hat, ist es sehr unwahrscheinlich, dass das beste Netz beim ersten Versuch einer zufälligen Hyperparameterwahl direkt gefunden wird. Aus diesem Grund verwendet man eine Hyperparametersuche, um die möglichst optimalen Hyperparameter für das Neuronale Netz finden zu können. Hierbei kamen konkret sogenannte *keras_tuner* zum Einsatz, welche Hyperparametersuche durchführen können. Dabei wurde *RandomSearch* verwendet, welche zufällige Hyperparameter auswählt, die im Suchraum sind und daraufhin dann eine bestimmte Anzahl an Netzen trainiert und miteinander vergleicht. Das Gleiche wurde mit *GridSearch* versucht. *GridSearch* hingegen versucht alle Kombinationen von Hyperparametern im Suchraum auszuprobieren und darauf Netze zu trainieren. Aufgrund der hohen Anzahl an verschiedenen Hyperparameterkombinationen war es jedoch nicht möglich, jede Kombination auszuprobieren. Daraufhin geben beide *Tuner* die beste Hyperparameterkombination aus, worauf man dann ein finales Neuronales Netz trainieren kann und die Leistungsfähigkeit miteinander vergleichen kann.

10.8 ROC Kurven

Eine ROC-Kurve ist eine gute Grafik, um die Leistungsfähigkeit von einem binären Klassifizierer zu überprüfen und diese miteinander vergleichen zu können. Auf der y-Achse plottet man die *Sensitivität* und auf der x-Achse *1 - Spezifität*. Um zu verstehen, wie die Kurve auf einer ROC-Kurve zustande kommt, schauen wir uns folgendes Beispiel an: [5]

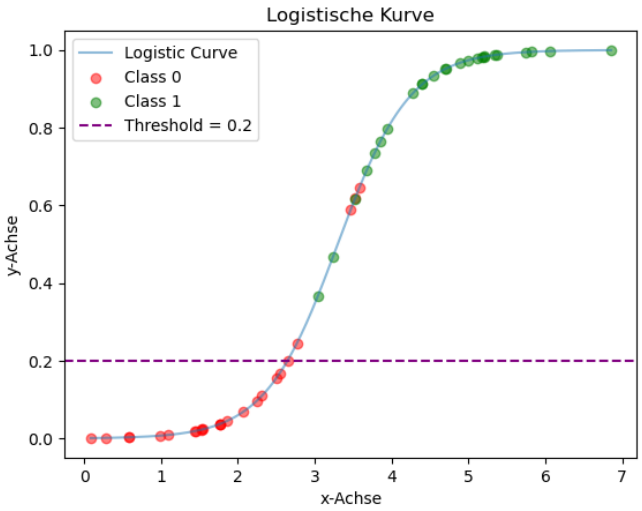

Abbildung 12: Beispielhafte Klassenaufteilung mit Threshold = 0.2
Quelle: Eigene Darstellung

Die roten Punkte gehören Klasse 0 und die grünen Punkte gehören Klasse 1 an. Dabei beschreibt der Threshold, hier bei 0.2, ab welchem y-Wert ein Punkt in die Klasse 1 klassifiziert wird. Formal also:

$$X = \begin{cases} 1, & falls\ X \geq T \\ 0, & falls\ X < T \end{cases}$$

Damit würde die Konfusionsmatrix für dieses konkrete Beispiel wie folgt aussehen:

	Vorhersage	
	0	1
Wirklich 0	20	5
Wirklich 1	0	25

Aus dieser *Konfusionsmatrix* lässt sich jetzt die *Spezifität* und *Sensitivität* errechnen, die man dann als einzelnen Punkt auf der *ROC-Kurve* abbilden kann. Nun kann aber ebenfalls ein anderer Threshold gewählt werden und dementsprechend würde sich eine andere Konfusionsmatrix und somit auch auf andere *Spezifität*- und *Sensitivitätswerte* ergeben. Für den Threshold=0.5 sieht das gegebene Beispiel dann folgendermaßen aus:

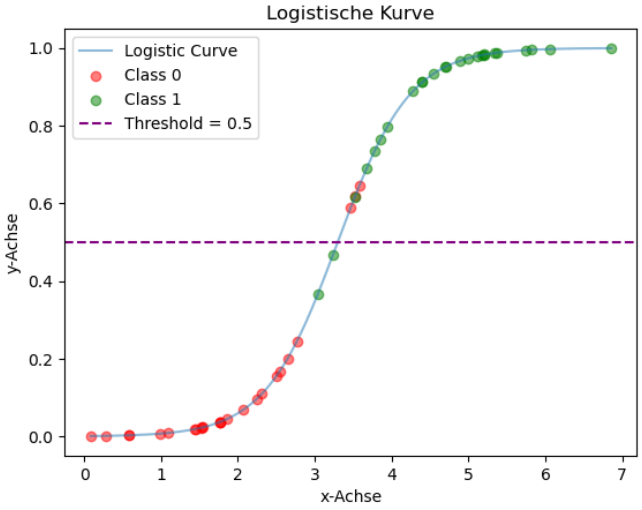

Abbildung 13: Beispielhafte Klassenaufteilung mit Threshold = 0.5
Quelle: Eigene Darstellung

Und die dazugehörige *Konfusionsmatrix*:

	Vorhersage	
	0	1
Wirklich 0	22	3
Wirklich 1	2	23

Für die *ROC-Kurve* werden eine gegebene Anzahl an Thresholds ausprobiert und daraus entstehen dann die Punkte im Graphen. Für das oben gegebene Beispiel könnte eine *ROC-Kurve* mit den Thresholds 0.2, 0.5 und 0.8 wie folgt aussehen:

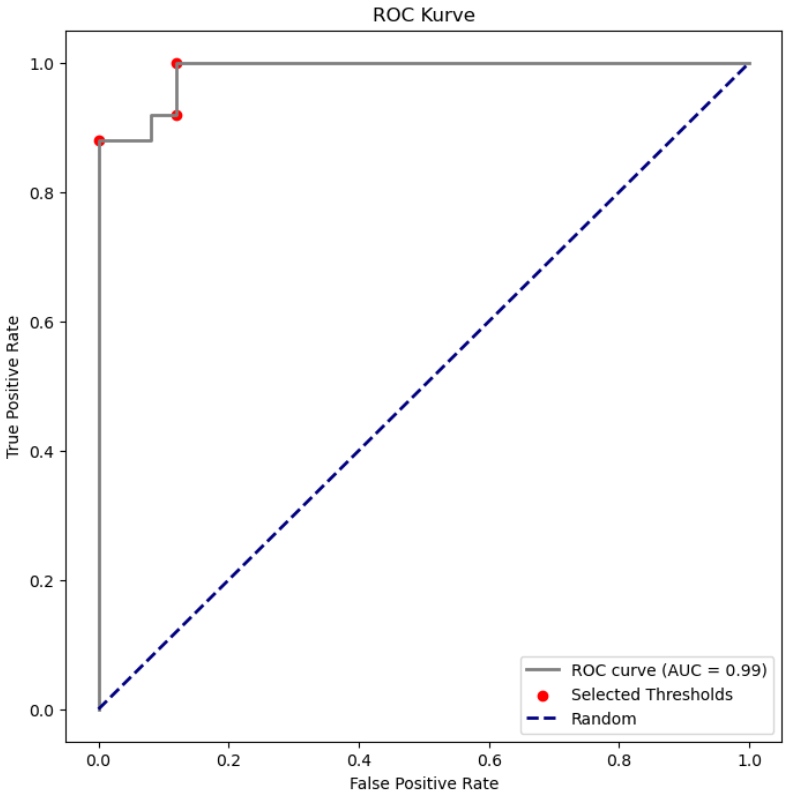

Abbildung 14: Beispiel einer ROC-Kurve
Quelle: Eigene Darstellung

Je näher diese Kurve sich dem Wert (0|1) annähert, desto besser schätzt man das gegebene Modell ein. Dabei beschreibt die Diagonale einen Klassifizierer, der zufällig entscheidet. Ein Modell sollte also in jedem Fall besser als ein Klassifizierer sein, der zufällig entscheidet. Als weitere Metrik bietet sich der sogenannte *AUC-Score (area under the curve)* an. Dieser beschreibt, wie groß der Anteil der Fläche unter der *ROC-Kurve* ist. Eine größere *area under the curve* deutet auf einen besseren Klassifizierer hin.

10.9 Leistungsfähigkeit des Netzes

10.9.1 Ergebnisse der Netze unterschiedlicher Klassenverteilung

Im bereits besprochenen Abschnitt **10.5** wurde darauf hingewiesen, dass möglicherweise Schwierigkeiten aufgrund der Klassenverteilung der Daten auftreten könnten. Allerdings konnte festgestellt werden, dass sich das Problem der Klassenverteilung nicht auf die Performance des Modells auswirkt, sondern lediglich auf die Lernfähigkeit des Modells.

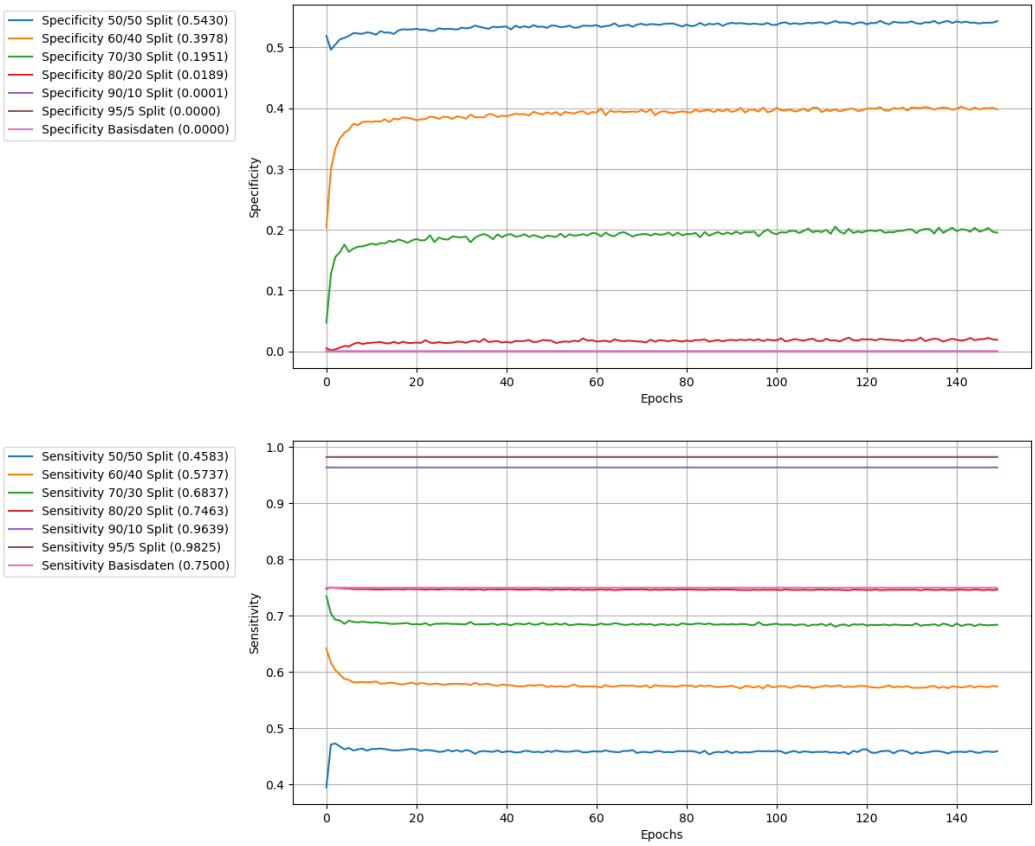

Abbildung 15: Spezifität und Sensitivität während des Trainings mit unterschiedlichen Klassenaufteilungen

Quelle: Eigene Darstellung, Datengrundlage: UK Biobank (2023)

Aus der Abbildung 15 geht hervor, je ausgeglichener die Klassenverteilung bei der Spezifität ist, desto mehr Schwankungen und Veränderungen während des Trainings zu sehen sind. Bei der Sensitivität sieht man eine ähnliche Tendenz nur umgekehrt, allerdings ist hier zu erwähnen, dass die Sensitivität der Basisdaten heraussticht. Würde man nun eine höhere Priorität auf die Sensitivität oder auf die Spezifität legen, könnte man das jeweilige Modell wählen, was in jener Metrik besser funktioniert. Da wir uns hier aber gleichermaßen für Sensitivität und Spezifität interessieren, ist die Wahl der Klassenverteilung für die Leistung des Modells unerheblich.

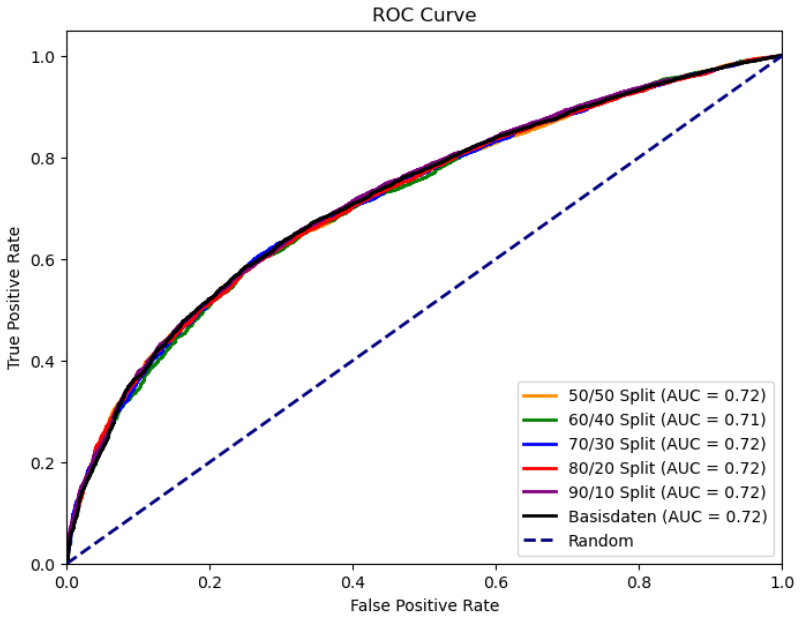

Abbildung 16: ROC-Kurve der unterschiedlichen Modelle mit variierenden Klassenverteilungen
Quelle: Eigene Darstellung, Datengrundlage: UK Biobank (2023)

Hier fällt auf, dass die Leistungsfähigkeit der Modelle nahezu gleich ist und tatsächlich die Klassenverteilung keine signifikante Auswirkung auf die Leistungsfähigkeit des Modells hat. Der beste ***AUC-Score*** **liegt hier bei 0.72**, was ein akzeptabler Wert ist, allerdings gibt es hier noch Raum für Verbesserung.

10.9.2 Leistungsfähigkeit des Netzes zwischen Männern und Frauen

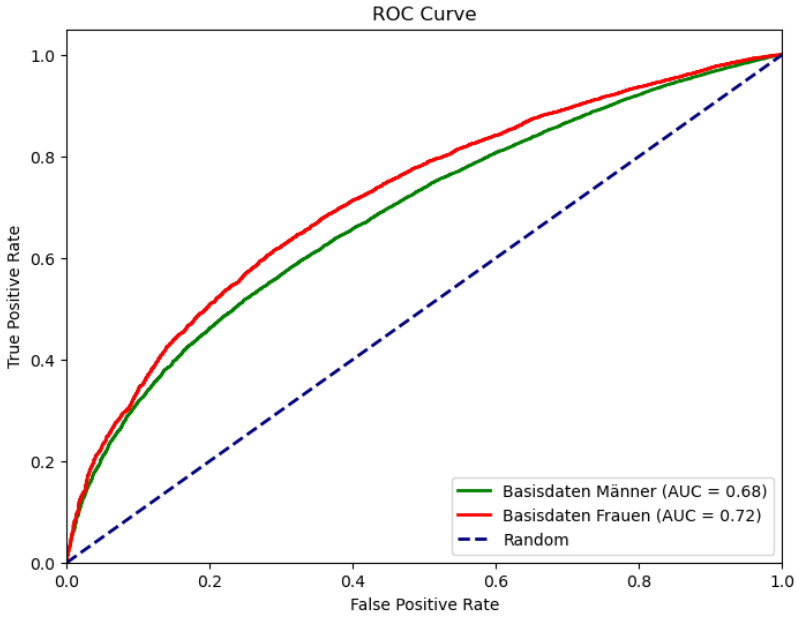

Abbildung 17: ROC-Kurve zum Vergleich der Leistungsfähigkeit zwischen Frauen und Männern
Quelle: Eigene Darstellung, Datengrundlage: UK Biobank (2023)

Hier fällt klar auf, dass die Aufteilung nach Geschlecht einen Unterschied in der Leistungsfähigkeit des Modells macht. **Das Modell der Frauen erreicht hier einen *AUC-Score* von 0.72**, während das **Modell der Männer nur einen *AUC-Score* von 0.68** erreicht.

10.9.3 Ergebnisse der Hyperparametersuche

Schlussendlich wurde, wie bereits in Kapitel **10.7** erwähnt, eine Hyperparametersuche durchgeführt, um die optimalen Hyperparameter zu finden und somit das Netz eventuell noch zu optimieren und bessere Leistungsfähigkeit zu erzielen. Dabei wurden insgesamt 6 Tuner mit einem *keras_tuner* durchgeführt, wovon 3 *RandomSearch Tuner* und 3 *GridSearch Tuner* waren. Diese hatten entweder 100 oder 500 Versuche, das gegebene Modell zu erstellen.
Wie bereits erwähnt, macht die Hyperparameterwahl bei den Basisdaten keinen Unterschied, da das Netz sich nicht verbessern kann. Deshalb wurde bei *GridSearch Modell 1 und 2* und *RandomSearch Modell 1 und 2* die 50/50 geteilten Daten verwendet. *GridSearch Modell 3* und *RandomSearch Modell 3* hingegen sind mit den Basisdaten trainiert worden. Der Suchbereich war folgender:

- num_layers [1;5] *Anzahl der Layer*
- units [32;512; step=32] *Anzahl der Neuronen pro Layer*
- activations [relu, sigmoid, leaky_relu, swish] *Aktivierungsfunktionen*
- use_dropout [1;0] *Benutzung von Dropout Ja/Nein*
- dropout_rate [0;0.5; step=0.1] *Dropoutwahrscheinlichkeit, falls use_dropout = 1*
- optimizer [adam, sgd, rmsprop, adagrad] *Optimierungsverfahren*
- learning_rate [$1e^{-4}$;$1e^{-1}$; log sampling] *Lernrate*

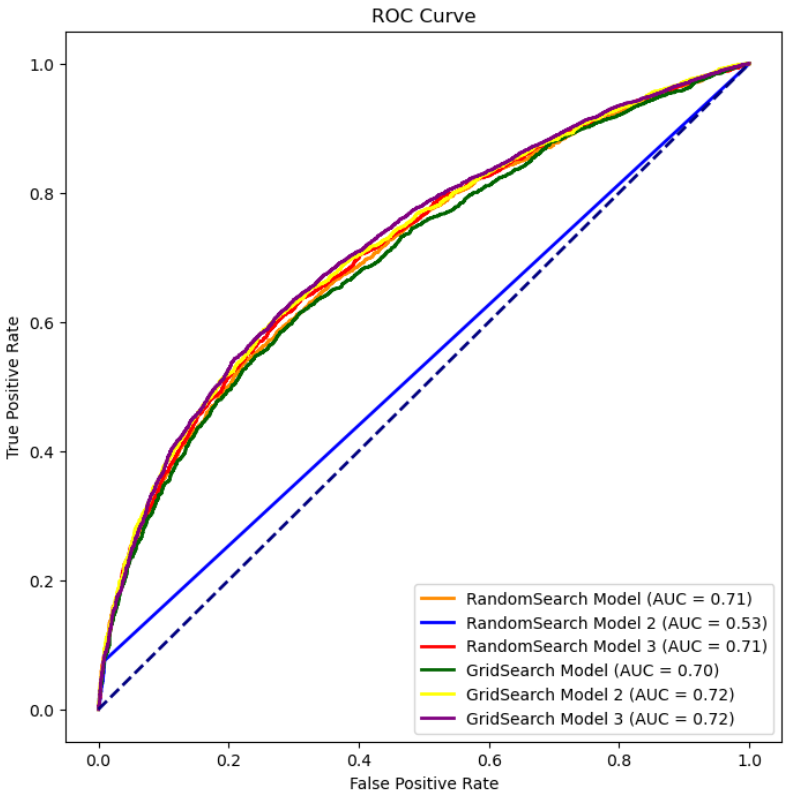

Abbildung 18: ROC-Kurve zum Vergleich der Leistungsfähigkeit unterschiedlicher Modelle mit Hyperparametersuche

Quelle: Eigene Darstellung, Datengrundlage: UK Biobank (2023)

Hier fällt nun auf, dass im Vergleich zum Basismodell, was in Kapitel **11.9.1** verwendet wurde, keine Verbesserung möglich war. Das *Random Search Modell 2* ist wohl ein Ausreißer, da die Suche der besten Hyperparameter zufällig verläuft. Das beste Modell hat die folgenden Hyperparameter:

- num_layers = 4 (2 Hidden Layers, 1 Input- und Outputlayer)
- units [9,128,128,1]
- activation: ReLU
- use_dropout = 0
- Aktivierung auf dem Outputlayer: Sigmoid
- optimizer: SGD (*stochastic gradient decent*)
- learning_rate = 0.01 (default)

11 Fazit

Das Ziel des Projektes war es, ein tieferes Verständnis für Neuronale Netze zu gewinnen. Dies konnte erreicht werden, indem ich mich zuerst mit den Grundlagen, wie etwa dem *künstlichen Neuron*, beschäftigt habe und davon ausgehend mein Wissen vertieft habe, indem ich beispielsweise ein tieferes Verständnis für die *Forward- und Backpropagation* gewinnen konnte. Hierbei war es zudem wichtig, die Mathematik hinter den Algorithmen zu verstehen. Dadurch konnte ich mir ein erstes Anwendungsbeispiel in Python anschauen und verstehen, wie es in Kapitel 9 anhand der MNIST Daten dargestellt wurde.
In Kapitel 10 wurde letztendlich die konkrete Anwendung an den UK Biobank Daten versucht. Dort mussten zuerst die Daten präprozessiert werden, um sie in das richtige Format für die Neuronalen Netze zu bekommen. Darauf wurde dann ein erstes Neuronales Netz trainiert. Nach der Feststellung, dass die Lernfähigkeit des Netzes eingeschränkt war, wurden einige Hyperparameterkombinationen ausprobiert, um das Problem weiter zu untersuchen. Dadurch wurde schnell klar, dass das Problem der Lernfähigkeit in der Klassenverteilung der Daten lag, woraufhin verschiedene Klassenverteilungen ausprobiert wurden, um dieses Phänomen weiter zu beobachten. Nachdem mehrere Netze mittels *keras_tuner* gebaut wurden, konnte man diese Netze in den *ROC-Kurven* miteinander vergleichen. Hieraus ging hervor, dass die Klassenverteilung keinen Einfluss auf die Leistungsfähigkeit des Netzes hatte und ein finaler AUC-Score von 0.72 erreicht werden konnte, was ein akzeptables Ergebnis ist.

12 Literatur- und Quellenverzeichnis

Literatur

[1] Chandra Sekhar, G., Mathai, A., Parkih, R. & Parkih, S. (2008) *Understanding and using sensivity, specificity and predictive values.*, Mumbai: Indian J Ophthalmol.

[2] Deisenroth, M., Faisal, A., Ong, C. (2020). *Mathematics for machine learning.*, Cambridge: Cambridge University Press.

[3] Drori, I. (2023). *The Science of Deep Learning.* Cambridge: Cambridge University Press.

[4] Ertel, W. (2021). *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung.* 5.Aufl., Wiesbaden: Springer Fachmedien Wiesbaden

[5] Fawcett, T. (2006). *An Introduction to ROC Analysis. In: Pattern Recognition Letters,* Volume 27, S.861-874

[6] Kuczyk, M. & von Klot, C. (2018). *Künstliche Intelligenz und neuronale Netze in der Urologie.* Berlin: Springer Medizin Verlag GmbH