

Bewegungserkennung in Kinect-Skelettdaten mithilfe von Faltungsnetzen im Vergleich zu Rekurrenten Neuronalen Netzen

Einleitung

Ziel dieser Projektarbeit ist das Erkennen mehrerer unterschiedlicher Bewegungsabläufe mithilfe von Faltungsnetzen. Hierbei wird das resultierende Modell direkt mit einem auf den gleichen Daten trainierten Rekurrenten Neuronalen Netz verglichen. Bei den verwendeten Daten handelt es sich um Skelettdaten einer Kinect v2. Konkret handelt es sich um den [NTU RGB+D 120 Action Recognition Datensatz](https://rose1.ntu.edu.sg/datasets/actionrecognition.asp) (<https://rose1.ntu.edu.sg/datasets/actionrecognition.asp>), der durch das ROSE Lab der Nanyang Technological University in Singapore zur Verfügung gestellt wurde. Untersucht wird ebenfalls der Einfluss von Daten Normalisierung und verschiedener Modellstrukturen auf die Vorhersagegenauigkeit.

Der Aktuelle Stand der Forschung

Die verwendeten Skelettdaten stehen als Sequenzen zur Verfügung. Somit ergeben sich 2 Eigenschaften der Daten, die berücksichtigt werden müssen. Zum einen die in den Skelettdaten hinterlegten Gelenkpunkte sowie deren Zusammenhänge. Zum anderen die zeitliche Abhängigkeit der dargestellten Aktion. Diese Eigenschaften haben in der Literatur zu 3 Ansätzen geführt. Zum einen lädt die Sequenzstruktur mit der zeitlichen Dimension zur Nutzung Rekurrenter Neuronaler Netze mit einer Gedächtnisfunktion ein, wie sie aus NLP Anwendungen bekannt ist. Das Gedächtnis kann man sich in diesem Anwendungsfall vielmehr "visuell" vorstellen. Bsp. hierfür wäre die Arbeit "[ACTION RECOGNITION USING VISUAL ATTENTION](https://arxiv.org/pdf/1511.04119v3.pdf)" von Shikhar Sharma, Ryan Kiros & Ruslan Salakhutdinov. Eine andere Herangehensweise wäre die Nutzung einer zeitlichen und räumlichen Faltung über die einzelnen Frames einer Sequenz hinweg. Die kann mit Faltungsnetzen aus der Bilderkennung verglichen werden. Hierbei wird jedoch nun über den Verlauf von Gelenkkordinaten statt über Farbkanäle und Pixelkoordinaten gefaltet. State of the art Modelle zur Bewegungsklassifizierung nutzen jedoch zusätzlich zu diesen beiden Faltungen eine weitere Eigenschaft der Skelettdaten. So werden die Skelette als Graphen interpretiert, welche die Zusammenhänge zwischen den einzelnen Gelenken beinhalten. So kann über eine Graphenfaltung die nicht nur die Bewegung der einzelnen Gelenke über Raum und Zeit berücksichtigt werden, sondern auch die Relation dieser Gelenke zueinander. Ein gutes Beispiel für die Verwendung dieser 3 Faltungen liefert das Paper [Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition](https://arxiv.org/pdf/1801.07455v2.pdf) (<https://arxiv.org/pdf/1801.07455v2.pdf>) von Sijie Yan, Yuanjun Xiong und Dahua Lin des *Department of Information Engineering der Chinese University of Hong Kong*. Zu guter Letzt kann ein solches Netz durch zusätzliche Attention Module, ähnlich wie sie in Rekurrenten Neuronalen Netzen vorkommen, erweitert werden. Die so entstehenden Netz Architekturen führen derzeit die Rangliste der Bewegungsklassifizierung an. Die aktuell besten Resultate liefert dabei im durch Lei Shi, Yifan Zhang, Jian Cheng und Hanqing Lu in 2019 veröffentlichte Paper [Two-Stream Adaptive Graph Convolutional Networks for Skeleton-Based Action Recognition](https://arxiv.org/pdf/1912.06971v1.pdf) (<https://arxiv.org/pdf/1912.06971v1.pdf>) vorgestellt wird. In dieser Arbeit werden wir uns aus Gründen der Komplexität auf die räumliche und zeitliche Faltung beschränken.

Der Datensatz

Bei dem verwendeten Datensatz handelt es sich um den [NTU RGB+D 120 Action Recognition Datensatz](https://rose1.ntu.edu.sg/datasets/actionrecognition.asp) (<https://rose1.ntu.edu.sg/datasets/actionrecognition.asp>), des ROSE Lab der Nanyang Technological University in Singapore. Der Datensatz umfasst 120 Klassen mit insgesamt 114,480 Sequenzen. Konkret handelt es sich um Aufnahmen verschiedener Länge welche mit unterschiedlichen Darstellern aus jeweils 3

verschiedenen Kamerawinkeln gleichzeitig aufgenommen wurden. Verwendet wurde dabei Kinect v2 Kameras. Enthalten sind für die Sequenzen sowohl RGB Videodaten als auch Tiefendaten, 3D Skelettdaten, und 3d infrarot Videos. die Auflösung der RGB Videos beträgt 1920x1080, für Tiefenbilder und Infrarot Videos 512x424. die Skelettdaten enthalten 3D koordinaten von 25 Gelenkpunkten in jedem Frame. Verwende werden in unserem Fall tatsächlich nur die Skelettdaten.

Die Aktionsklassen

Die Bewegungen, auf sich für diese Projektarbeit geeinigt wurde, sind die folgenden:

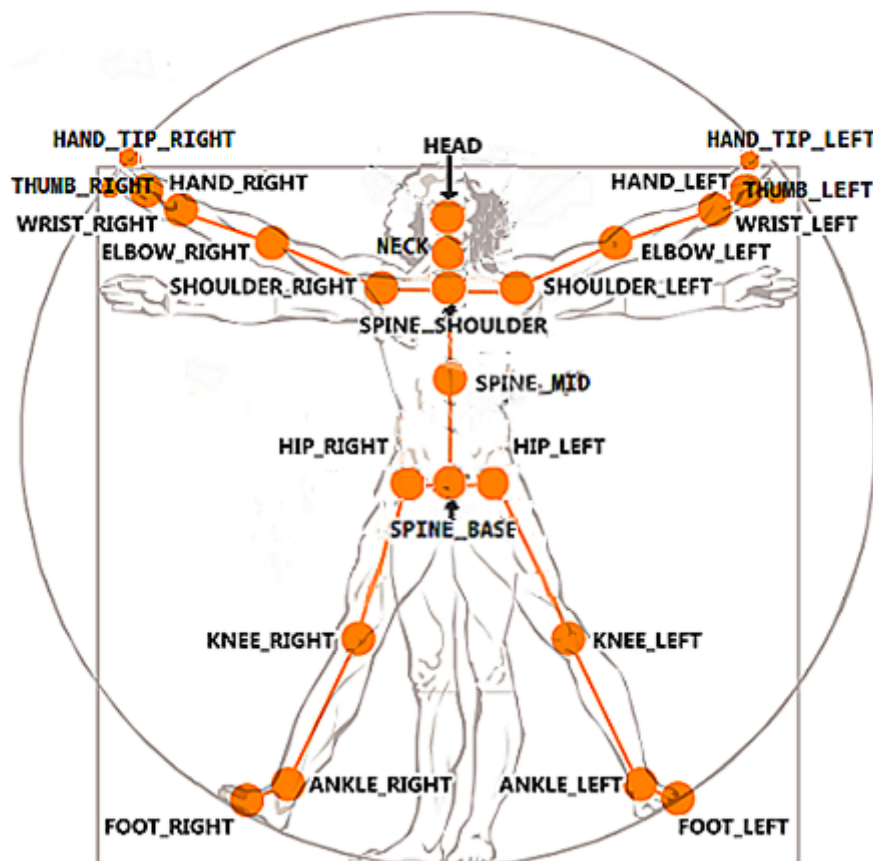
- Spielen auf dem smartphone
- Aufnehmen eines Selfies
- Telephonieren
- Eine Restklasse mit sonstigen Bewegungen

Die Restklasse setzt sich zu gleichen Teilena aus Sequenzen der folgenden Klassen zusammen:

- Wasser trinken
- Zähne putzen
- Klatschen
- Jacke anziehen
- Brille ausziehen
- Hut/Mütze absetzen

Aufbau der Skelettdaten

Die 25 Gelenkpunkte, welche die Skelettdaten ausmachen, sind jeweils mit X,Y und Z Werten versehen und stellen folgende Körperpunkte dar:



Die Nummerierung der Keypoints ist die folgende:

SPINEBASE	0
SPINEMID	1
NECK	2
HEAD	3
SHOULDERLEFT	4
ELBOWLEFT	5
WRISTLEFT	6
HANDTIPLEFT	7
SHOULDERRIGHT	8
ELBOWRIGHT	9
WRISTRIGHT	10
HANDTIPRIGHT	11
HIPLEFT	12
KNEELEFT	13
ANKLELEFT	14
FOOTLEFT	15
HIPRIGHT	16
KNEERIGHT	17
ANKLERIGHT	18
FOOTRIGHT	19
SPINESHOULDER	20
HANDTIPLEFT	21
THUMBLEFT	22
HANDTIPRIGHT	23
THUMBRIGHT	24

Setup und Abhängigkeiten

Imports von Abhängigkeiten sowie Verbindung mit google Drive

In [1]:

```
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

In [2]:

```
!rm -r /content/sample_data
```

In [3]:

```

%load_ext tensorboard
%matplotlib inline

from tensorflow import keras
import pickle
import os
import tensorflow as tf
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, roc_auc
from keras.utils import to_categorical
import numpy as np
from keras.preprocessing.sequence import pad_sequences
import logging
import traceback
from sklearn.model_selection import train_test_split
import random
import time
import multiprocessing as mp
from math import sin, cos, sqrt

import math
import numpy as np
import time
import random
import matplotlib.pyplot as plt
import importlib

from IPython.display import HTML

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3
import matplotlib.animation as animation

print(tf.__version__)
print("Environment Ready")

```

2.3.0

Environment Ready

In [4]:

```

import keras.backend as K
K.clear_session()

```

In [5]:

```

bones = np.array(((0,1),(0,12),(0,16),(1,20),(20,2),(20,4),(20,8),(2,3),(4,5),(5,6),(6,7),(
    (10,11),(11,23),(11,24),(12,13),(13,14),(14,15),(16,17),(17,18),(18,19)))

anim_body_parts = ['HipM', 'Spine', 'Neck', 'Head', 'ShoulderR', 'ElbowR', 'WristR', 'HandR',
    'ShoulderL', 'ElbowL', 'WristL', 'HandL', 'HipR', 'KneeR', 'AnkleR', 'FootR',
    'HipL', 'KneeL', 'AnkleL', 'FootL', 'Breast', 'PointerR', 'PinkyR', 'PointerL',
    'PinkyL']
target_names = ['phone call', 'play with phone', 'taking a selfie', 'rest class']

```

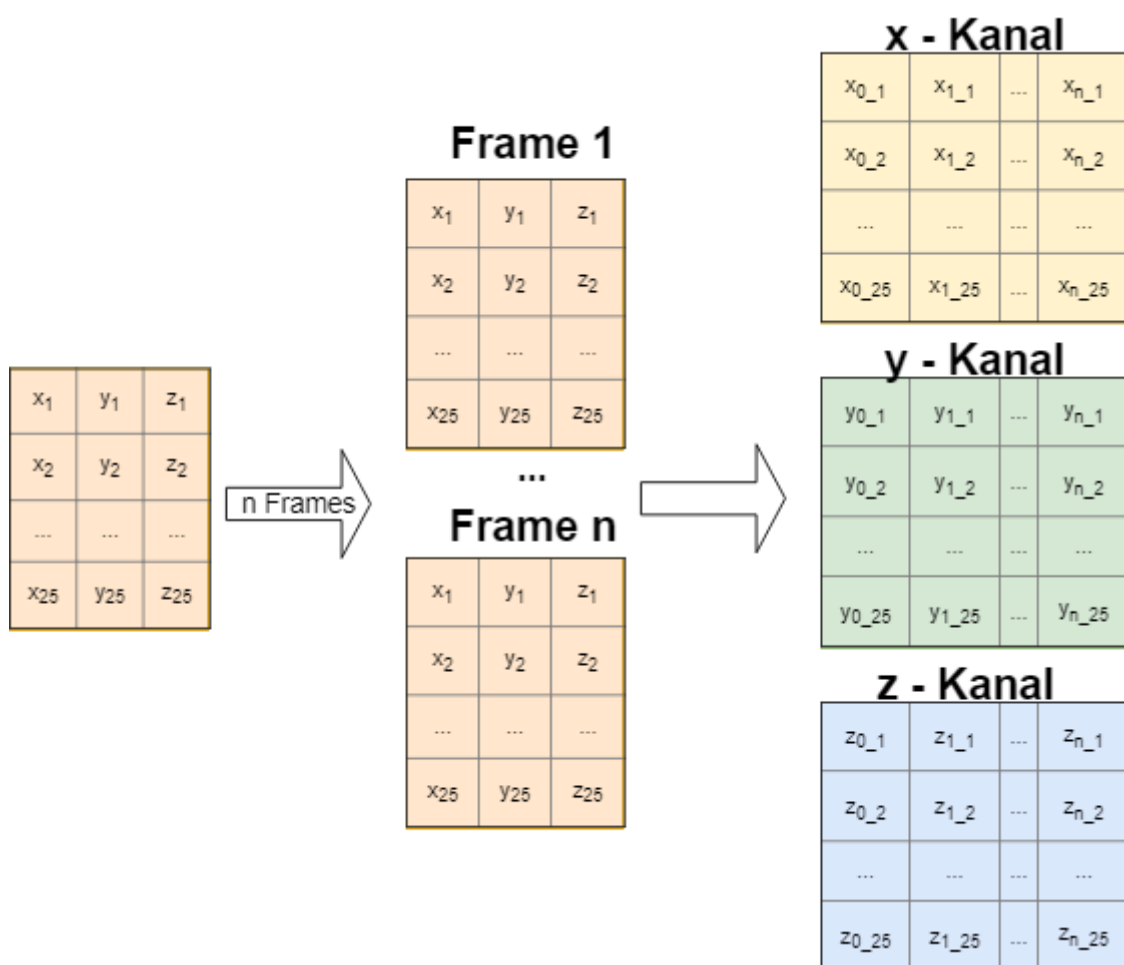
Datenaufbereitung

Die Datenaufbereitung können wir in folgende Schritte einteilen:

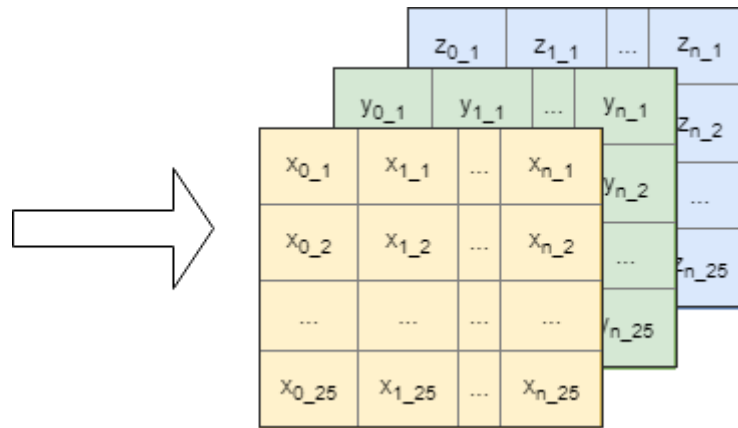
1. Festlegen der Datenstruktur
2. Labeln der Daten
3. Normalisierung der Daten
4. Padding der Daten
5. Split der Trainings und Testdaten

Die Datenstruktur

Unsere Daten liegen als in Aktionsklassen unterteilte Sequenzen vor. Jede Sequenz kann dabei in der Länge an Frames variieren. Ein einzelnes Frame enthält, wie bereits in der einleitung zum Datenset erläutert, 25 Gelenkpunkte mit jeweils 3 Koordinaten. Die Überlegte Datenstruktur sieht nun folgendermaßen aus:



Analog zu Bilddaten mit Kanälen für Rot, Grün und Blau stellen wir unsere Frames so auf, dass wir Kanäle für die X, Y und Z Koordinaten erhalten. Diese legen wir dann übereinander:



Die so entstehende Datenstruktur hat dann je nach Sequenzlänge eine dimensionalität von $(n, 25, 3)$, wobei n unsere Sequenzlänge darstellt.

Labeln der Daten

Da unsere Daten in Aktionsklassen unterteilt vorliegen, wird jeder Aktionsklasse ein Index zugeteilt.

Normalisierung der Daten

Für unser Training unterscheiden wir 4 Fälle der Normalisierung:

- Unnormalisierte Daten, also keine Normalisierung
- Normalisierung in ein Körperkoordinatensystem.

Hierbei wir anhand der Hüfte und der Wirbelsäule des Skelettes ein Koordinatensystem aufgebaut. Alle Gelenkpunkt-Koordinaten werden in dieses System übertragen. dies wird für jedes Frame wiederholt.

- Normalisierung in ein Körperkoordinatensystem relativ zu erstem Sequenzframe. Der unterschied zur vorherigen Normalisierung besteht darin, dass das Koordinatensystem nicht für jedes Frame neu aufgestellt wird, sondern nur im ersten Frame.
- Rotationsnormalisierung relativ zum ersten Sequenzframe. Ähnlich zu letzten Normalisierung, mit einer zusätzlichen Rotation des Skelettes. Hierbei wird das Skelett so rotiert, dass der X-Vektor auf einer Linie mit der Schulterlinie des Skelettes liegt. Dies ist in der Visualisierung der Normalisierungen klar erkennbar. Als Vorlage diente die im bereits erwähnten durch Lei Shi, Yifan Zhang, Jian Cheng und Hanqing veröffentlichten [Paper \(https://arxiv.org/pdf/1912.06971v1.pdf\)](https://arxiv.org/pdf/1912.06971v1.pdf) verwendete Normalisierung.

Die Implementierung der verschiedenen Normalisierungen sind in den folgenden Codeblöcken zu sehen.

Padding der Sequenzen

Da unsere Sequenzen unterschiedliche Längen haben, stehen wir vor einem Problem. Für ein Faltungsnetz müssten alle unsere Sequenzen entweder die gleiche Länge haben, oder wir müssten die Sequenzen in Abschnitte gleicher Länge einteilen. Das Zurechtschneiden der Sequenzen hat jedoch einige Nachteile:

- Überlappen sich unsere Sequenz-Abschnitte, wenn wir zum Beispiel mit einem Sliding Window Verfahren arbeiten, so werden sich einige Sequenzabschnitte beim Training oder der Validierung wiederholen. Dies wäre nicht optimal.
- Zerschneiden wir eine längere Sequenz in kürzere Abschnitte, so werden einige dieser Abschnitte wenig oder gar keine Information zu der Klasse enthalten, die sie darstellen sollen. Wegwerfen kann man sie auch nicht, da nicht klar ist, welcher der erstellten Abschnitte wichtige Informationen enthält und welcher nicht.

die einzig verbleibende Option wäre also, die Sequenzen alle auf eine Länge zu bringen. Wie genau dies umgesetzt werden kann, ohne die Modelle beim trainieren zu beeinträchtigen, lässt sich aus dem Paper [EFFECTS OF PADDING ON LSTMS AND CNNs \(https://arxiv.org/pdf/1903.07288.pdf\)](https://arxiv.org/pdf/1903.07288.pdf) von Dwarampudi Mahidhar Reddy und Subba Reddy herauslesen. Demnach hat das Padding auf Faltungsnetze keinerlei Einfluss. RNN's hingegen sind sehr anfällig gegenüber der Art des Paddings. Aus diesem Grund haben wir uns für ein Pre-Padding/Pre-Trimming entschieden, bei dem wir auf eine Länge von 170 Frames aufstocken oder runterkürzen. Die Begründung hierfür ist die mittlere Länge der Sequenzen in unseren ausgewählten Aktionsklassen. Dass vorne abgeschnitten wird statt hinten, lässt damit rechtfertigen, dass die Sequenzen eher im Mittleren bis hinteren Bereich ausgeübt werden.

Hilfsmethoden zur Darstellung der Skelettdaten nach Normalisierung

In []:

```
def plot_frame(data, equal_axis, is_3d):
    fig = plt.figure(figsize=(8,4))
    if is_3d:
        ax = fig.add_subplot(111, projection='3d')
    else:
        ax = fig.add_subplot(111)

    X=selected_seq_x[10][:,0]
    Y=selected_seq_x[10][:,1]
    Z=selected_seq_x[10][:,2]

    if is_3d:
        ax.scatter(X,Y,Z, marker='o')#, c=confidence, cmap='cividis_r')
        for i, part in zip(range(len(X)), anim_body_parts):
            ax.text(X[i], Y[i], Z[i], (' '+str(i)+' ') +part) , fontsize=5)
        for con in bones:
            ax.plot([X[con[0]],X[con[1]]],[Y[con[0]],Y[con[1]]],[Z[con[0]],Z[con[1]]], color='k')
        ax.set_xlabel('X-Axis')
        ax.set_ylabel('Y-Axis')
        ax.set_zlabel('Z-Axis')
    else:
        ax.scatter(X,Y, marker='o')#, c=confidence, cmap='cividis_r')
        for i, part in zip(range(len(X)), anim_body_parts):
            ax.text(X[i], Y[i], (str(i)+' ') +part) , fontsize=6)
        for con in bones:
            ax.plot([X[con[0]],X[con[1]]],[Y[con[0]],Y[con[1]]], color='k')
        ax.set_xlabel('X-Axis')
        ax.set_ylabel('Y-Axis')

    # make scale correct
    if equal_axis:
        max_range = np.array([X.max()-X.min(), Y.max()-Y.min(), Z.max()-Z.min()]).max() / 2

        mid_x = (X.max()+X.min()) * 0.5
        mid_y = (Y.max()+Y.min()) * 0.5
        mid_z = (Z.max()+Z.min()) * 0.5
        ax.set_xlim(mid_x - max_range, mid_x + max_range)
        ax.set_ylim(mid_y - max_range, mid_y + max_range)
        if is_3d:
            ax.set_zlim(mid_z - max_range, mid_z + max_range)
```

In []:

```

def make_3D_axis_equal(ax, X,Y,Z):
    max_range = np.array([X.max()-X.min(), Y.max()-Y.min(), Z.max()-Z.min()]).max() / 2.0

    mid_x = (X.max()+X.min()) * 0.5
    mid_y = (Y.max()+Y.min()) * 0.5
    mid_z = (Z.max()+Z.min()) * 0.5
    ax.set_xlim(mid_x - max_range, mid_x + max_range)
    ax.set_ylim(mid_y - max_range, mid_y + max_range)
    ax.set_zlim(mid_z - max_range, mid_z + max_range)

def animate_scatters(iteration, data, scatters, connections):
    scatters._offsets3d = (data[iteration][:,0], data[iteration][:,1], data[iteration][:,2])

    for con, i in zip(bones,range(len(bones))):
        # xy and z are set seperatly
        connections[i].set_data(
            [data[iteration][:,0][con[0]],data[iteration][:,0][con[1]]],
            [data[iteration][:,1][con[0]],data[iteration][:,1][con[1]]])

        connections[i].set_3d_properties(
            [data[iteration][:,2][con[0]],data[iteration][:,2][con[1]]])

    return scatters, connections

def main(data,class_label, save=False):

    # Attaching 3D axis to the figure
    fig = plt.figure()
    ax = p3.Axes3D(fig)

    # Initialize plots
    scatters = ax.scatter(data[0][:,0],data[0][:,1],data[0][:,2])
    make_3D_axis_equal(ax, data[0][:,0],data[0][:,1],data[0][:,2])

    connections = [ax.plot(
        [data[0][:,0][con[0]],data[0][:,0][con[1]]],
        [data[0][:,1][con[0]],data[0][:,1][con[1]]],
        [data[0][:,2][con[0]],data[0][:,2][con[1]]], color='k')[0]
        for con in bones]

    # Number of iterations
    iterations = len(data)

    # Setting the axes properties
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    ax.set_title(f'Animated Skeleton Sequence, Class: {target_names[class_label]}')

    # Provide starting angle for the view.
    ax.view_init(25, 10)
    ani = animation.FuncAnimation(fig, animate_scatters, iterations, fargs=(data, scatters,
        interval=33, blit=False, repeat=True)

    # 33 miliseconds of delay between each frame results in 30fps

    if save:
        Writer = animation.writers['ffmpeg']
        writer = Writer(fps=30, metadata=dict(artist='Me'), bitrate=1800, extra_args=['-vco

```



```
ani.save('3d-scatted-animated.mp4', writer=writer)

plt.show()
return ani
```

Methoden zu Normalisierung

Rotationsnormalisierung

Umsetzung der Rotationsnormalisierung.

In []:

```

body_parts = ['Nose', 'Neck', 'RShoulder', 'RElbow', 'RWrist', 'LShoulder', 'LElbow', 'LWrist', 'RH
              , 'RAnkle', 'LHip', 'LKnee', 'LAnkle', 'REye', 'LEye', 'REar', 'LEar', 'Smartphone']
# ----- Imported Methods -----
def angle_between(v1, v2):
    """ Returns the angle in radians between vectors 'v1' and 'v2':
        >>> angle_between((1, 0, 0), (0, 1, 0))
        1.5707963267948966
        >>> angle_between((1, 0, 0), (1, 0, 0))
        0.0
        >>> angle_between((1, 0, 0), (-1, 0, 0))
        3.141592653589793
    """
    if np.abs(v1).sum() < 1e-6 or np.abs(v2).sum() < 1e-6:
        return 0
    v1_u = unit_vector(v1)
    v2_u = unit_vector(v2)
    return np.arccos(np.clip(np.dot(v1_u, v2_u), -1.0, 1.0))
def unit_vector(vector):
    """ Returns the unit vector of the vector. """
    return vector / np.linalg.norm(vector)
def rotation_matrix(axis, theta):
    """
    Return the rotation matrix associated with counterclockwise rotation about
    the given axis by theta radians.
    """
    if np.abs(axis).sum() < 1e-6 or np.abs(theta) < 1e-6:
        return np.eye(3)
    axis = np.asarray(axis)
    axis = axis / sqrt(np.dot(axis, axis))
    a = cos(theta / 2.0)
    b, c, d = -axis * sin(theta / 2.0)
    aa, bb, cc, dd = a * a, b * b, c * c, d * d
    bc, ad, ac, ab, bd, cd = b * c, a * d, a * c, a * b, b * d, c * d
    return np.array([[aa + bb - cc - dd, 2 * (bc + ad), 2 * (bd - ac)],
                    [2 * (bc - ad), aa + cc - bb - dd, 2 * (cd + ab)],
                    [2 * (bd + ac), 2 * (cd - ab), aa + dd - bb - cc]])
# source: https://github.com/Lshiwjx/2s-AGCN/blob/master/data_gen/rotation.py
# normalize data
def normalize_keypoints(frame, norm_frame):
    frame = frame.copy()
    # minus spine
    body_center = norm_frame[1,:].copy()
    for i in range(len(frame)):
        frame[i] = frame[i,:] - body_center
    #make x-axis parallel to shoulders
    joint_lshoulder = norm_frame[4,:].copy()
    joint_rshoulder = norm_frame[8,:].copy()
    axis_x = np.cross(joint_rshoulder - joint_lshoulder, [1, 0, 0])
    angle_x = angle_between(joint_rshoulder - joint_lshoulder, [1, 0, 0])
    matrix_x = rotation_matrix(axis_x, angle_x)
    for i in range(len(frame)):
        frame[i] = np.dot(matrix_x, frame[i,:])
    # make z-axis parallel to spine
    joint_hip = norm_frame[0,:].copy()
    joint_neck = norm_frame[1,:].copy()
    axis_z = np.cross(joint_neck - joint_hip, [0, 0, 1])
    angle_z = angle_between(joint_neck - joint_hip, [0, 0, 1])
    matrix_z = rotation_matrix(axis_z, angle_z)
    for i in range(len(frame)):

```

```

        frame[i] = np.dot(matrix_z, frame[i,:])
    return frame

def normalize_sequence(sequence):
    # choose which frame is basis for normalization
    # basis = sequence[int(len(sequence)//2)] # 7 ~ middle of 15 frames (length of one sequence)
    basis = sequence[0]
    norm_sequence = np.array([normalize_keypoints(frame, basis) for frame in sequence])
    return norm_sequence

def rotation_normalization(data):
    export_data = []
    for sequence in data:
        export_data.append(normalize_sequence(sequence))
    return np.asarray(export_data)

```

Körperkoordinaten Normalisierung

Umsetzung der Normalisierung in ein eigenes Körperkoordinatensystem. Enthält 3 nennenswerte Methoden:

- **normalize:** Jedes Frame wird individuell normalisiert. Der Körperschwerpunkt wird in jedem Frame auf den Punkt (0,0,0) gesetzt.
- **advanced_normalization:** Der Körperschwerpunkt wird nur im ersten Frame auf (0,0,0) gesetzt. Die Koordinaten in den folgenden Frames werden in das gleiche Koordinatensystem umgewandelt. So wird eine Bewegung der Person im Raum theoretisch besser erhalten.
- **normalizeToTfPoseModel:** Wie normalize, allerdings werden Gelenkpunkte weggelassen und ihre Ordnung umgewandelt, um dem Tf-Pose Gelenkmodell zu entsprechen. So könnte eine Erkennung auf Live-Videos mit dem Tf-Pose Keypoint Detektor umgesetzt werden.

In []:

```

def transformToBodyCoords(rotation_matrix, base_keypoint, coord):
    rc = np.array(coord)
    hc = np.array(base_keypoint)
    vector = rotation_matrix.dot(rc - hc)
    return vector

def getVector(coord_1, coord_2):
    a = np.array(coord_1)
    b = np.array(coord_2)
    return b - a

def getBodyBaseVectors(keypoints, baseKeypoints):
    base_vector_x_norm = [0, 0, 0]
    base_vector_y_norm = [0, 0, 0]
    base_vector_x = getVector(baseKeypoints[0], baseKeypoints[1])
    base_vector_y = getVector(baseKeypoints[0], baseKeypoints[2])
    if np.linalg.norm(base_vector_x) != 0:
        base_vector_x_norm = base_vector_x / np.linalg.norm(base_vector_x)

    if np.linalg.norm(base_vector_y) != 0:
        base_vector_y_norm = base_vector_y / np.linalg.norm(base_vector_y)

    base_vector_z_norm = np.cross(base_vector_x_norm, base_vector_y_norm)

    return base_vector_x_norm, base_vector_y_norm, base_vector_z_norm

def advanced_normalization(data):
    export_data = []
    for sequence in data:
        export_sequence = []
        sequence_startframe = sequence[0]
        rotation_matrix = np.column_stack(getBodyBaseVectors(sequence_startframe, [sequence_startframe + 1, sequence_startframe + 2, sequence_startframe + 3]))
        for frame in sequence:
            export_keypoints = []
            for keypoint in range(len(frame)):
                export_keypoints.append(transformToBodyCoords(rotation_matrix, sequence_startframe + keypoint, frame[keypoint]))
            export_sequence.append(np.asarray(export_keypoints, dtype=np.float32))
        export_data.append(np.asarray(export_sequence))
    return np.asarray(export_data)

def normalize(data):
    export_data = []
    for sequence in data:
        export_sequence = []
        for frame in sequence:
            export_keypoints = []
            rotation_matrix = np.column_stack(getBodyBaseVectors(frame[0], [frame[0], frame[12], frame[13]]))
            for keypoint in range(len(frame)):
                export_keypoints.append(transformToBodyCoords(rotation_matrix, frame[0], frame[keypoint]))
            export_sequence.append(np.asarray(export_keypoints, dtype=np.float32))
        export_data.append(np.asarray(export_sequence))
    return np.asarray(export_data)

def normalizeToTfPoseModel(frame):
    export_keypoints = []
    #order is important to match keypoint order of tf-pose model
    relevant_keypoints = [0,3,20,8,9,10,4,5,6,16,17,18,12,13,14]
    for i in relevant_keypoints:

```

```

        export_keypoints.append(frame[i])
    rotation_matrix = np.column_stack(getBodyBaseVectors(export_keypoints, [export_keypoints[
    for keypoint in range(len(export_keypoints)):
        export_keypoints[keypoint] = transformToBodyCoords(rotation_matrix, frame[0], export_
    return np.asarray(export_keypoints, dtype=np.float32)

```

Allgemeine Hilfsmethoden zur vorbereitung der Trainingsdaten

In []:

```

def add_class_label(data):
    new_data = []
    for class_id in range(len(data)):
        subset = np.asarray(data[class_id])
        new_subset = []
        for sequence_id in range(len(subset)):
            new_subset.append(np.array([np.asarray(subset[sequence_id]), class_id]))
        new_data.append(np.asarray(new_subset))
    return np.asarray(new_data)

def prepare_training_data(data):
    padded_X, y = prepare_data_without_split(data)
    return train_test_split(padded_X, y, test_size=0.2, random_state = 42, stratify = y)

def prepare_data_without_split(data):
    X = []
    for i in range(len(data)):
        X.append(np.asarray(data[i][0]))
    y = data[:,1]
    y = to_categorical(y)
    padded_X = pad_sequences(X, maxlen=170, dtype = 'float32', padding="pre", truncating="pre")
    return padded_X, y

```

Trainingsdaten Pipeline

In []:

```

def preparation_pipeline(normalization):
    data = np.load('/content/drive/My Drive/Studium/Projektarbeit/data/extra_action_classes.n

    if normalization is None:
        labeled_data = add_class_label(data)
        flattened_data = np.concatenate([labeled_data[0], labeled_data[1], labeled_data[2], lab
        X_train, X_test, y_train, y_test = prepare_training_data(flattened_data)
        name_prefix = 'unnormalized_'
    else:
        p = mp.Pool(4)

        if normalization == 'normal' :
            normalized_data = p.map(normalize,[data[0], data[1], data[2], data[3]])
            normalized_labeled_data = add_class_label(normalized_data)
            normalized_flattened_data = np.concatenate([np.asarray(normalized_labeled_data[0]),
                                                         np.asarray(normalized_labeled_data[1]),
                                                         np.asarray(normalized_labeled_data[2]),
                                                         np.asarray(normalized_labeled_data[3])])
            X_train, X_test, y_train, y_test = prepare_training_data(normalized_flattened_data)
            name_prefix = 'normalized_'

        elif normalization == 'advanced':
            advanced_normalized_data = p.map(advanced_normalization,[data[0], data[1], data[2], d
            advanced_normalized_labeled_data = add_class_label(advanced_normalized_data)
            advanced_normalized_flattened_data = np.concatenate([np.asarray(advanced_normalized_l
                                                         np.asarray(advanced_normalized_l
                                                         np.asarray(advanced_normalized_l
                                                         np.asarray(advanced_normalized_l

            X_train, X_test, y_train, y_test = prepare_training_data(advanced_normalized_flattene
            name_prefix = 'advanced_normalized_'

        elif normalization == 'rotation':
            rotation_normalized_data = p.map(rotation_normalization,[data[0], data[1], data[2], d
            rotation_normalized_labeled_data = add_class_label(rotation_normalized_data)
            rotation_normalized_flattened_data = np.concatenate([np.asarray(rotation_normalized_l
                                                         np.asarray(rotation_normalized_l
                                                         np.asarray(rotation_normalized_l
                                                         np.asarray(rotation_normalized_l

            X_train, X_test, y_train, y_test = prepare_training_data(rotation_normalized_flattene
            name_prefix = 'rotation_normalized_'

    return X_train, X_test, y_train, y_test, name_prefix

```

Visualisierung der Normalisierung

In []:

```

from matplotlib import rc
rc('animation', html='jshtml')

```

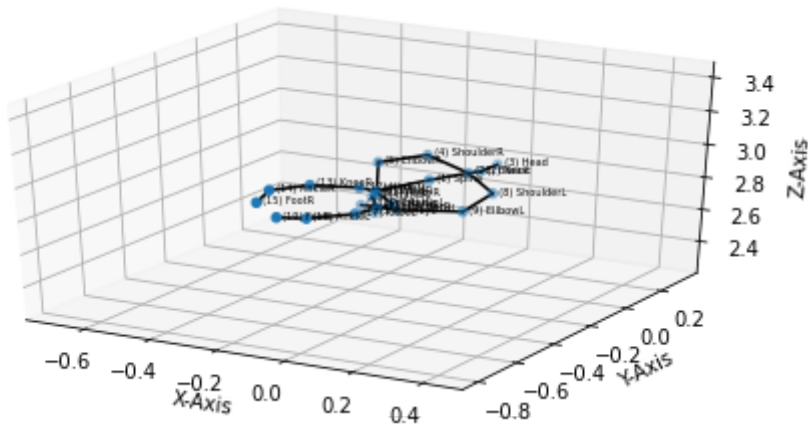
Für unnormalisierte Daten

In []:

```

X_train, X_test, y_train, y_test, name_prefix = preparation_pipeline(None)
selected_seq_x = X_train[0]
selected_seq_y = np.argmax(y_train[0])
selected_seq_x = np.array([frame for frame in selected_seq_x if frame.sum()!=0])
plot_frame(selected_seq_x[10], equal_axis=True, is_3d=True)

```



Als Animation:

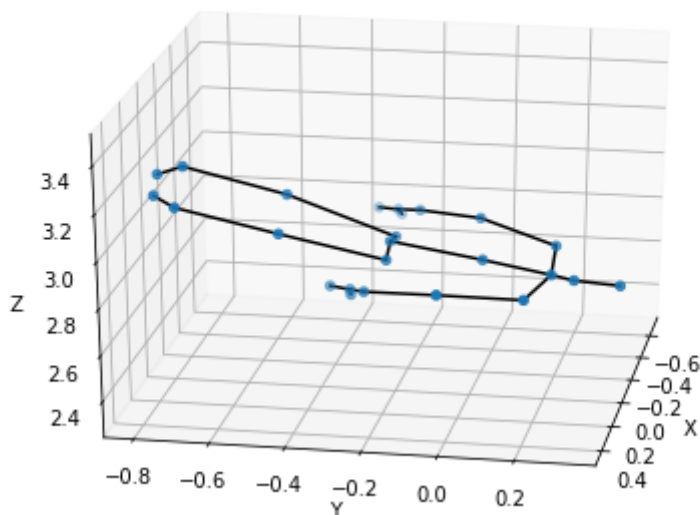
In []:

```

ani = main(selected_seq_x, selected_seq_y, save=False)
ani

```

Animated Skeleton Sequence, Class: play with phone



Out[14]:

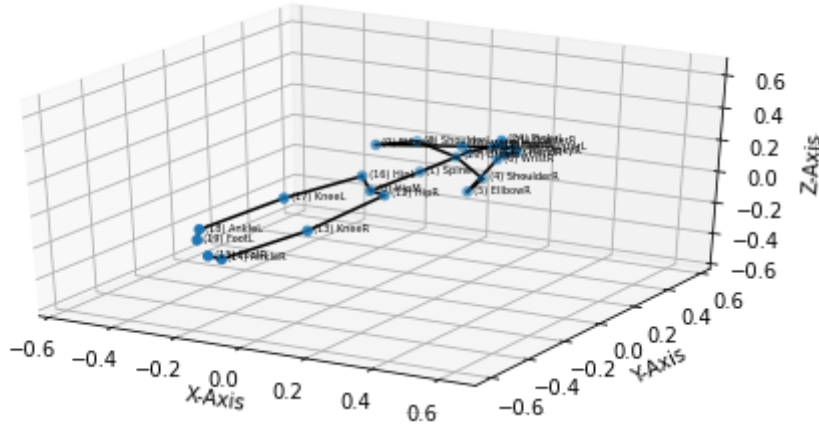


☐ Once
 ☒ Loop
 ☐ Reflect

für Normalisierte Daten

In []:

```
X_train, X_test, y_train, y_test, name_prefix = preparation_pipeline('normal')
selected_seq_x = X_train[0]
selected_seq_y = np.argmax(y_train[0])
selected_seq_x = np.array([frame for frame in selected_seq_x if frame.sum()!=0])
plot_frame(selected_seq_x[10], equal_axis=True, is_3d=True)
```

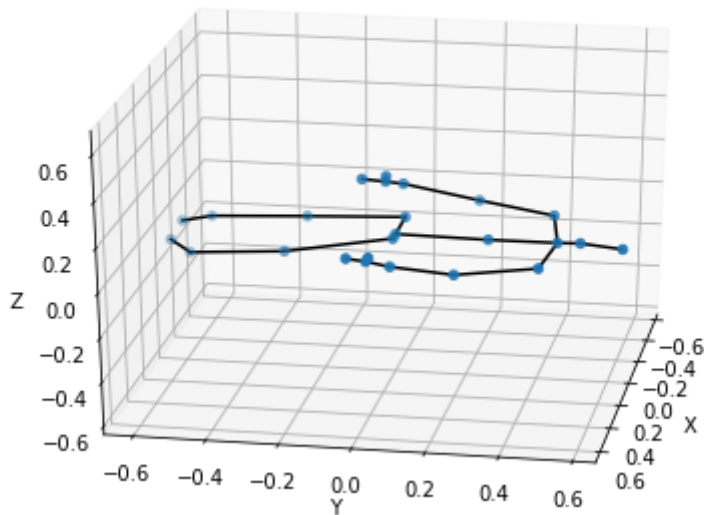


Als Animation:

In []:

```
ani = main(selected_seq_x, selected_seq_y, save=False)
ani
```

Animated Skeleton Sequence, Class: play with phone



Out[16]:



In []:

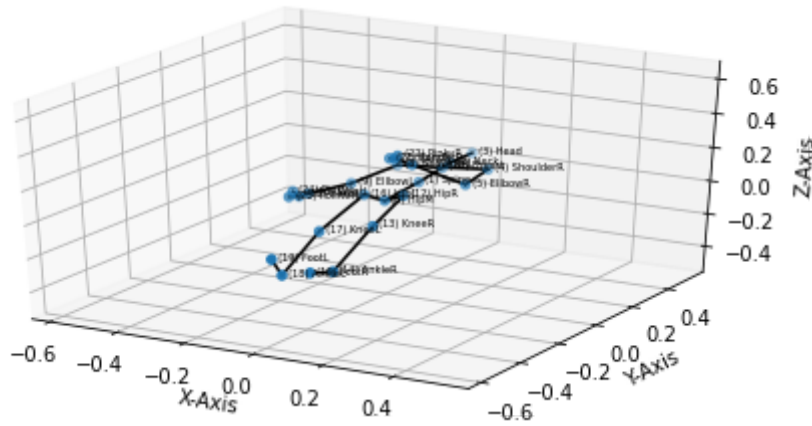
für Advanced Normalized Daten

In []:

```

X_train, X_test, y_train, y_test, name_prefix = preparation_pipeline('advanced')
selected_seq_x = X_train[120]
selected_seq_y = np.argmax(y_train[120])
selected_seq_x = np.array([frame for frame in selected_seq_x if frame.sum()!=0])
plot_frame(selected_seq_x[10], equal_axis=True, is_3d=True)

```



Als Animation:

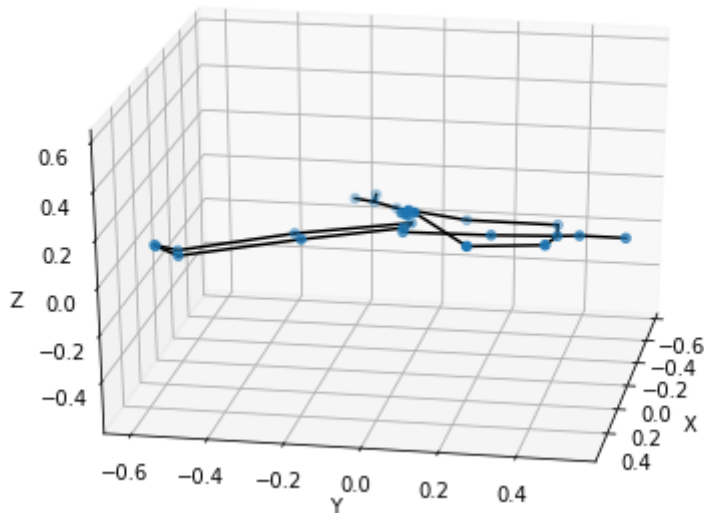
In []:

```

ani = main(selected_seq_x, selected_seq_y, save=False)
ani

```

Animated Skeleton Sequence, Class: rest class



Out[18]:



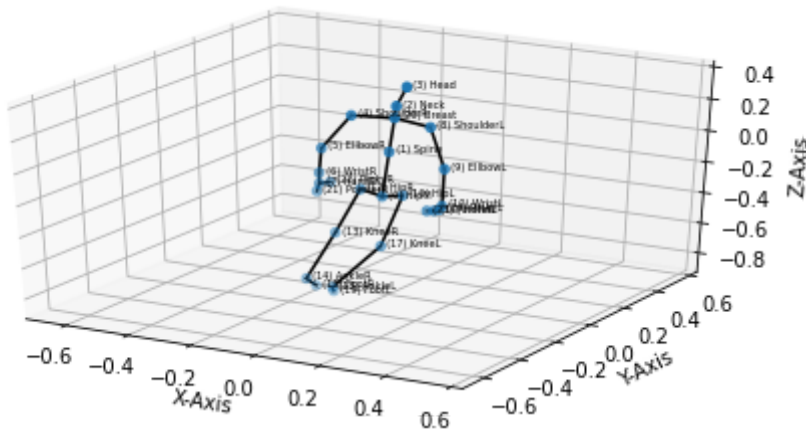
für Rotations Normalisierte Daten

In []:

```

X_train, X_test, y_train, y_test, name_prefix = preparation_pipeline('rotation')
selected_seq_x = X_train[300]
selected_seq_y = np.argmax(y_train[300])
selected_seq_x = np.array([frame for frame in selected_seq_x if frame.sum()!=0])
plot_frame(selected_seq_x[10], equal_axis=True, is_3d=True)

```



Als Animation

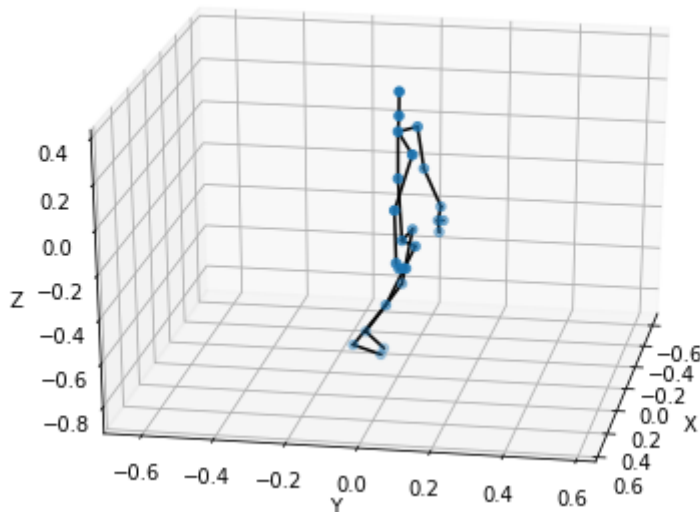
In []:

```

ani = main(selected_seq_x, selected_seq_y, save=False)
ani

```

Animated Skeleton Sequence, Class: taking a selfie



Out[20]:



☐ Once
 ☒ Loop
 ☐ Reflect

Cross View und Cross Subject Trainingsdaten

Die bisherigen Aufbereitungen der Trainingsdaten berücksichtigen keinen speziellen Anwendungsfall. Generell kann bei dem Training mit unseren Daten jedoch unter zwei Fällen unterschieden werden; Cross View und Cross Subject.

Cross View

Cross view ist allgemein der leichtere der beiden Fälle. Hierbei wird der Trainings und Validierungs-Split so ausgesucht, dass unser Modell auf zwei der drei Kameraansichten trainiert wird. Bei der Validierung muss das Modell nun die Bewegung aus der dritten Kameraansicht erkennen können.

Cross Subject

Cross Subject ist wiederum schwieriger. Hierbei wird sichergestellt, dass im Trainings und Validierungsdatensatz die Aktionen/Bewegungen von unterschiedlichen Darstellern ausgeübt werden. so soll sichergestellt werden, dass unser Modell gut generalisieren kann, und sich nicht an eigenheiten unserer Darsteller orientiert. Dieser Fall ist bei weitem schwieriger zu behandeln.

Wir werden bei der Evaluierung unserer Modelle diese beiden Fälle ebenfalls berücksichtigen und unser Modell separat mit entsprechenden Daten trainieren und auswerten.

Erstellen der Cross View / Cross Subject Trainingsdaten

Das Datenset wird mit vordefinierten Einteilungen für Cross View und Cross Subject zur Verfügung gestellt. Einzig die Restklasse muss noch zusammengestellt werden. Je nach Verwendungsbedarf müssen in den folgenden Blöcken entsprechende Codestellen auskommentiert / reaktiviert werden

In []:

```

with open("/content/drive/My Drive/Studium/Projektarbeit/data/action_classes_multi_body_cvc
specialized_data = pickle.load(pickle_file)
A001_sequences = specialized_data['001'] # drink water
A003_sequences = specialized_data['003'] # brush teeth
A010_sequences = specialized_data['010'] # clapping
A014_sequences = specialized_data['014'] # put on jacket
A019_sequences = specialized_data['019'] # take of glasses
A021_sequences = specialized_data['021'] # take off a hat/cap

A028_sequences = specialized_data['028']
A029_sequences = specialized_data['029']
A032_sequences = specialized_data['032']

# combine to one class
extra_class_cross_view_train = []
extra_class_cross_view_test = []
extra_class_cross_subject_train = []
extra_class_cross_subject_test = []

for action_class in [A001_sequences, A003_sequences, A010_sequences, A014_sequences, A019
    for sample in random.sample(list(action_class['cross_view_train']), 112):
        extra_class_cross_view_train.append(sample[:, :, :3] )

    for sample in random.sample(list(action_class['cross_subject_train']), 112):
        extra_class_cross_subject_train.append(sample[:, :, :3] )

    for sample in random.sample(list(action_class['cross_view_val']), 46):
        extra_class_cross_view_test.append(sample[:, :, :3] )

    for sample in random.sample(list(action_class['cross_subject_val']), 46):
        extra_class_cross_subject_test.append(sample[:, :, :3] )

cross_view_train = np.array([np.array([sample[:, :, :3] for sample in A028_sequences['cro
    np.array([sample[:, :, :3] for sample in A029_sequences['cro
    np.array([sample[:, :, :3] for sample in A032_sequences['cro
    extra_class_cross_view_train])
cross_view_test = np.array([np.array([sample[:, :, :3] for sample in A028_sequences['cros
    np.array([sample[:, :, :3] for sample in A029_sequences['cros
    np.array([sample[:, :, :3] for sample in A032_sequences['cros
    extra_class_cross_view_test])
cross_subject_train = np.array([np.array([sample[:, :, :3] for sample in A028_sequences['
    np.array([sample[:, :, :3] for sample in A029_sequences['
    np.array([sample[:, :, :3] for sample in A032_sequences['
    extra_class_cross_subject_train])
cross_subject_test = np.array([np.array([sample[:, :, :3] for sample in A028_sequences['c
    np.array([sample[:, :, :3] for sample in A029_sequences['cr
    np.array([sample[:, :, :3] for sample in A032_sequences['cr
    extra_class_cross_subject_test])

np.save('/content/drive/My Drive/Studium/Projektarbeit/data/cross_view_train', cross_view_t
np.save('/content/drive/My Drive/Studium/Projektarbeit/data/cross_view_test', cross_view_te
np.save('/content/drive/My Drive/Studium/Projektarbeit/data/cross_subject_train', cross_sub
np.save('/content/drive/My Drive/Studium/Projektarbeit/data/cross_subject_test', cross_subj

```

Laden der Datensätze, falls sie bereits erstellt wurden:

In []:

```
cross_view_train = np.load('/content/drive/My Drive/Studium/Projektarbeit/data/cross_view_t  
cross_view_test = np.load('/content/drive/My Drive/Studium/Projektarbeit/data/cross_view_te  
cross_subject_train = np.load('/content/drive/My Drive/Studium/Projektarbeit/data/cross_sub  
cross_subject_test = np.load('/content/drive/My Drive/Studium/Projektarbeit/data/cross_subj
```

Normalisieren der Daten:

In []:

```

data = cross_view_train
p = mp.Pool(4)
normalized_data = p.map(rotation_normalization,[data[0], data[1], data[2], data[3]])
normalized_labeled_data = add_class_label(normalized_data)
normalized_flattened_data = np.concatenate([np.asarray(normalized_labeled_data[0]),
                                             np.asarray(normalized_labeled_data[1]),
                                             np.asarray(normalized_labeled_data[2]),
                                             np.asarray(normalized_labeled_data[3])])

np.random.shuffle(normalized_flattened_data)
cross_view_train_X, cross_view_train_y = prepare_data_without_split(normalized_flattened_data)

data = cross_view_test
p = mp.Pool(4)
normalized_data = p.map(rotation_normalization,[data[0], data[1], data[2], data[3]])
normalized_labeled_data = add_class_label(normalized_data)
normalized_flattened_data = np.concatenate([np.asarray(normalized_labeled_data[0]),
                                             np.asarray(normalized_labeled_data[1]),
                                             np.asarray(normalized_labeled_data[2]),
                                             np.asarray(normalized_labeled_data[3])])

np.random.shuffle(normalized_flattened_data)
cross_view_test_X, cross_view_test_y = prepare_data_without_split(normalized_flattened_data)

data = cross_subject_train
p = mp.Pool(4)
normalized_data = p.map(rotation_normalization,[data[0], data[1], data[2], data[3]])
normalized_labeled_data = add_class_label(normalized_data)
normalized_flattened_data = np.concatenate([np.asarray(normalized_labeled_data[0]),
                                             np.asarray(normalized_labeled_data[1]),
                                             np.asarray(normalized_labeled_data[2]),
                                             np.asarray(normalized_labeled_data[3])])

np.random.shuffle(normalized_flattened_data)
cross_subject_train_X, cross_subject_train_y = prepare_data_without_split(normalized_flattened_data)

data = cross_subject_test
p = mp.Pool(4)
normalized_data = p.map(rotation_normalization,[data[0], data[1], data[2], data[3]])
normalized_labeled_data = add_class_label(normalized_data)
normalized_flattened_data = np.concatenate([np.asarray(normalized_labeled_data[0]),
                                             np.asarray(normalized_labeled_data[1]),
                                             np.asarray(normalized_labeled_data[2]),
                                             np.asarray(normalized_labeled_data[3])])

np.random.shuffle(normalized_flattened_data)
cross_subject_test_X, cross_subject_test_y = prepare_data_without_split(normalized_flattened_data)

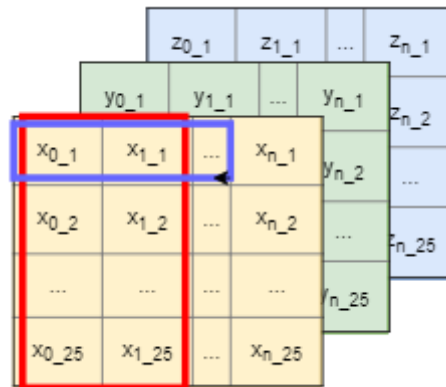
```

Die Modelle

Bei der Architektur der Faltungsnetze werden 4 verschiedene Ansätze verfolgt. Diese unterscheiden sich in erster Linie in der Form und Größe der Faltungskerne sowie in der Anzahl der Knoten im Fully Connected Layer.

Die verschiedenen Faltungskerne

Die Daten werden dem Modell in der bereits erläuterten Form übergeben. Diese erlaubt uns nun mehrere Möglichkeiten bei der Wahl der Faltungskerne. Allerdings gilt es in diesem Fall eine Besonderheit zu beachten. Während bei der Bildverarbeitung eine Faltung über die verschiedenen Farbkanäle durchaus sinnvoll macht, wäre dies in unserem Fall eher bedenklich. Eine Faltung über die räumlichen Koordinaten X, Y und Z eines Gelenkpunktes scheint wenig sinnvoll. Eine Faltung über die Koordinaten eines Gelenkespuunktes über mehrere Frames hinweg hingegen schon. Der in blauer Farbe markierte Faltungskern würde zum Beispiel über mehrere Frames hinweg die X Koordinaten falten, dann die Y und die Z Koordinaten. Der Rote Faltungskern würde sich nicht nur einen Gelenkpunkt beschränken sondern die Faltung über die Frames hinweg für alle Gelenkpunkte gleichzeitig durchführen.



Eine Faltung über mehrere Frames hinweg stellt eine Faltung über die Zeit hinweg, also eine Zeitfaltung. Da in unserem Fall keine Graphendarstellung der Gelenkpunkte vorliegt, ist eine Faltung einer kleinen Gruppe von Gelenkpunkten über die Zeit eher suboptimal, da der genaue Zusammenhang der Gelenke zueinander in den Daten nicht vorliegt. Deshalb haben wir nur die Wahl zwischen diesen beiden Arten von Kernen.

1. wir Falten jedes Gelenk individuell über die Zeit
2. Wir Falten alle Gelenke zusammen über die Zeit

Welchen Faltungskern wir wählen, hängt stark von unserer Bewegung ab. Sind bei einer Aktion alle Gelenkpunkte in Bewegung, so eignet sich eine Faltung über alle Punkte. Bewegt sich hingegen nur ein einziger Gelenkpunkt, so ist die Faltung über ein einziges Gelenk vorzuziehen.

Relevant ist auch die Anzahl der Faltungskerne. Sobald wir eine hohe Anzahl an Kernen voraussetzen liefert die Faltung über alle Gelenkpunkte theoretisch vergleichbare Resultate, jedoch mit dem Preis einer höheren Parameteranzahl.

Modell Architektur

Nachdem die Frage der Faltungskerne behandelt wurde, steht nun der Aufbau einer Netzarchitektur an. Hierbei wurden 4 mögliche Architekturen ausprobiert. Bei allen Netzen wird eine Glorot Normalverteilung zur Initialisierung der Parameter gewählt, sowie eine RELU als Aktivierungsfunktion. Als Optimizer dient der Adam Optimizer mit einem Cross Entropy Loss.

Modellentwurf 1

Das erste Modell setzt sich aus 2 Zeitfaltungen sowie 2 Pooling Layern zusammen. Die Faltung besteht dabei jeweils aus Zeitpfaltungen über einzelne Gelenkpunkte. Auch das Max Pooling wird nur auf individuellen Gelenken durchgeführt. Die Faltung findet jeweils über 5 Frames statt. Der Stride der Faltungs und Pooling Kerne liegt bei 1. Auf das Faltungsnetz folgen 4 fully connected Layer mit Dropout Schichten.

Netz Zusammenfassung:

Layer (type)	Output Shape	Param #
conv2d_20 (Conv2D)	(None, 166, 25, 32)	512
max_pooling2d_10 (MaxPooling)	(None, 162, 25, 32)	0
conv2d_21 (Conv2D)	(None, 158, 25, 32)	5152
max_pooling2d_11 (MaxPooling)	(None, 154, 25, 32)	0
flatten_8 (Flatten)	(None, 123200)	0
dense_36 (Dense)	(None, 100)	12320100
dropout_12 (Dropout)	(None, 100)	0
dense_37 (Dense)	(None, 50)	5050
dropout_13 (Dropout)	multiple	0
dense_38 (Dense)	(None, 20)	1020
dense_39 (Dense)	(None, 20)	420
dense_40 (Dense)	(None, 4)	84
Total params: 12,332,338		
Trainable params: 12,332,338		
Non-trainable params: 0		

In []:

```
def define_and_compile_model(lr = 0.001):
    model = tf.keras.Sequential()
    input_conv_layer = tf.keras.layers.Conv2D(
        input_shape = (170,25,3),
        data_format = 'channels_last',
        kernel_size = (5,1),
        strides = 1,
        padding = 'valid',
        activation = tf.keras.activations.relu,
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        filters = 32
    )
    pooling_layer_1 = tf.keras.layers.MaxPool2D(
        data_format = 'channels_last',
        pool_size = (5,1),
        strides = 1,
        padding = 'valid'
    )
    conv_layer_2 = tf.keras.layers.Conv2D(
        data_format = 'channels_last',
        kernel_size = (5,1),
        strides = 1,
        padding = 'valid',
        activation = tf.keras.activations.relu,
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        filters = 32
    )

    pooling_layer_2 = tf.keras.layers.MaxPool2D(
        data_format = 'channels_last',
        pool_size = (5,1),
        strides = 1,
        padding = 'valid'
    )

    conv2_flat = tf.keras.layers.Flatten()

    dense_layer_1 = tf.keras.layers.Dense(
        activation = tf.keras.activations.relu,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        units = 100
    )

    dense_layer_2 = tf.keras.layers.Dense(
        activation = tf.keras.activations.relu,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        units = 50
    )

    dense_layer_3 = tf.keras.layers.Dense(
        activation = tf.keras.activations.relu,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        units = 20
    )

    dense_layer_4 = tf.keras.layers.Dense(
```

```

        activation = tf.keras.activations.relu,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        units = 20
    )
dropout_layer_hard = tf.keras.layers.Dropout(.35)

dropout_layer = tf.keras.layers.Dropout(.25)

dense_layer_output = tf.keras.layers.Dense(
    units = 4,
    bias_initializer = tf.keras.initializers.glorot_normal(),
    kernel_initializer = tf.keras.initializers.glorot_normal(),
    activation = tf.keras.activations.softmax
)

model.add(input_conv_layer)
model.add(pooling_layer_1)
model.add(conv_layer_2)
model.add(pooling_layer_2)
model.add(conv2_flat)
model.add(dense_layer_1)
model.add(dropout_layer_hard)
model.add(dense_layer_2)
model.add(dropout_layer)
model.add(dense_layer_3)
model.add(dropout_layer)
model.add(dense_layer_4)
model.add(dropout_layer)
model.add(dense_layer_output)
model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = lr), loss=tf.keras.l
return model

```

Modellentwurf 2

Das zweite Modell setzt sich aus 3 Zeitfaltungen und Max-Pooling Layern zusammen. Die Faltungen und das Maxpooling finden wieder nur auf individuellen Gelenken statt. allerdings wurde die Faltung in der 2ten und 3ten Faltungsschicht von 5 auf 3 Frames reduziert. Die Pooling Layer wurden ebenfalls auf eine Breite von 2 Frames reduziert. Der Stride bleibt für die Faltung und das Pooling identisch zum ersten Modell.

Netz Zusammenfassung:

Layer (type)	Output Shape	Param #
conv2d_22 (Conv2D)	(None, 166, 25, 32)	512
max_pooling2d_12 (MaxPooling)	(None, 165, 25, 32)	0
conv2d_23 (Conv2D)	(None, 163, 25, 32)	3104
max_pooling2d_13 (MaxPooling)	(None, 162, 25, 32)	0
conv2d_24 (Conv2D)	(None, 160, 25, 32)	3104
max_pooling2d_14 (MaxPooling)	(None, 159, 25, 32)	0
flatten_9 (Flatten)	(None, 127200)	0
dense_41 (Dense)	(None, 100)	12720100
dropout_14 (Dropout)	(None, 100)	0
dense_42 (Dense)	(None, 50)	5050
dropout_15 (Dropout)	multiple	0
dense_43 (Dense)	(None, 20)	1020
dense_44 (Dense)	(None, 4)	84
Total params: 12,732,974		
Trainable params: 12,732,974		
Non-trainable params: 0		

In []:

```
def define_and_compile_second_model(lr=0.001):
    model = tf.keras.Sequential()
    input_conv_layer = tf.keras.layers.Conv2D(
        input_shape = (170,25,3),
        data_format = 'channels_last',
        kernel_size = (5,1),
        strides = 1,
        padding = 'valid',
        activation = tf.keras.activations.relu,
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        filters = 32
    )

    pooling_layer_1 = tf.keras.layers.MaxPool2D(
        data_format = 'channels_last',
        pool_size = (2,1),
        strides = 1,
        padding = 'valid'
    )

    conv_layer_2 = tf.keras.layers.Conv2D(
        data_format = 'channels_last',
        kernel_size = (3,1),
        strides = 1,
        padding = 'valid',
        activation = tf.keras.activations.relu,
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        filters = 32
    )

    pooling_layer_2 = tf.keras.layers.MaxPool2D(
        data_format = 'channels_last',
        pool_size = (2,1),
        strides = 1,
        padding = 'valid'
    )

    conv_layer_3 = tf.keras.layers.Conv2D(
        data_format = 'channels_last',
        kernel_size = (3,1),
        strides = 1,
        padding = 'valid',
        activation = tf.keras.activations.relu,
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        filters = 32
    )

    pooling_layer_3 = tf.keras.layers.MaxPool2D(
        data_format = 'channels_last',
        pool_size = (2,1),
        strides = 1,
        padding = 'valid'
    )

    conv2_flat = tf.keras.layers.Flatten()

    dense_layer_1 = tf.keras.layers.Dense(
        activation = tf.keras.activations.relu,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        units = 100
    )
```

```

dense_layer_2 = tf.keras.layers.Dense(
    activation = tf.keras.activations.relu,
    bias_initializer = tf.keras.initializers.glorot_normal(),
    kernel_initializer = tf.keras.initializers.glorot_normal(),
    units = 50
)

dense_layer_3 = tf.keras.layers.Dense(
    activation = tf.keras.activations.relu,
    bias_initializer = tf.keras.initializers.glorot_normal(),
    kernel_initializer = tf.keras.initializers.glorot_normal(),
    units = 20
)

dropout_layer_hard = tf.keras.layers.Dropout(.35)
dropout_layer = tf.keras.layers.Dropout(.2)

dense_layer_output = tf.keras.layers.Dense(
    units = 4,
    bias_initializer = tf.keras.initializers.glorot_normal(),
    kernel_initializer = tf.keras.initializers.glorot_normal(),
    activation = tf.keras.activations.softmax
)

model.add(input_conv_layer)
model.add(pooling_layer_1)
model.add(conv_layer_2)
model.add(pooling_layer_2)
model.add(conv_layer_3)
model.add(pooling_layer_3)
model.add(conv2_flat)
model.add(dense_layer_1)
model.add(dropout_layer_hard)
model.add(dense_layer_2)
model.add(dropout_layer)
model.add(dense_layer_3)
model.add(dropout_layer)
model.add(dense_layer_output)
model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = lr), loss=tf.keras.l
return model

```

Modellentwurf 3

Das dritte Modell stellt einen kompletten Entwurfswechsel dar. die Faltungsschichten werden auf 2 reduziert:

1. Der erste Faltungskern wird auf eine Breite von 10 Frames erhöht und faltet weiterhin auf individuellen Gelenken.
2. Der Zweite Faltungskern faltet nun hingegen über alle Gelenke gleichzeitig. Er bekommt also eine Höhe von 25. Seine Breite wird auf 5 gesetzt, es wird also über 5 Frames gefaltet.

Auf Pooling Layer wird komplett verzichtet. Stattdessen wird der Stride für die Faltungskerne über die Frames hinweg auf 2 erhöht.

Die Idee dahinter ist simpel:

- Die erste Faltungsschicht soll für jedes Gelenk bestimmte Bewegungsmuster extrahieren.
- Die zweite Schicht soll über diese alle diese Muster hinweg über die Zeit falten um die Gesamte Bewegung des Körpers zu deuten.

Dies sorgt für eine verringerte Anzahl an Parametern in den Faltungsschichten. Auffällig in den vorherigen Modellen war die viel zu hohe Anzahl an Parametern, ausgelöst durch den zu großen Anteil an fully connected Layern. Dieser Anteil wird nun stark reduziert, da die Ausgabe der letzten Faltungsschicht um ein vielfaches verkleinert wurde.

Netz Zusammenfassung:

Layer (type)	Output Shape	Param #
conv2d_25 (Conv2D)	(None, 81, 25, 32)	992
conv2d_26 (Conv2D)	(None, 39, 1, 32)	128032
flatten_10 (Flatten)	(None, 1248)	0
dense_45 (Dense)	(None, 100)	124900
dropout_16 (Dropout)	multiple	0
dense_46 (Dense)	(None, 50)	5050
dense_47 (Dense)	(None, 20)	1020
dense_48 (Dense)	(None, 20)	420
dense_49 (Dense)	(None, 4)	84
Total params: 260,498		
Trainable params: 260,498		
Non-trainable params: 0		

In []:

```
def define_and_compile_third_model(lr=0.001):
    model = tf.keras.Sequential()
    input_conv_layer = tf.keras.layers.Conv2D(
        input_shape = (170,25,3),
        data_format = 'channels_last',
        kernel_size = (10,1),
        strides = (2,1),
        padding = 'valid',
        activation = tf.keras.activations.relu,
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        filters = 32
    )

    conv_layer_2 = tf.keras.layers.Conv2D(
        data_format = 'channels_last',
        kernel_size = (5,25),
        strides = 2,
        padding = 'valid',
        activation = tf.keras.activations.relu,
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        filters = 32
    )

    conv2_flat = tf.keras.layers.Flatten()

    dense_layer_1 = tf.keras.layers.Dense(
        activation = tf.keras.activations.relu,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        units = 100
    )

    dense_layer_2 = tf.keras.layers.Dense(
        activation = tf.keras.activations.relu,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        units = 50
    )

    dense_layer_3 = tf.keras.layers.Dense(
        activation = tf.keras.activations.relu,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        units = 20
    )

    dense_layer_4 = tf.keras.layers.Dense(
        activation = tf.keras.activations.relu,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        units = 20
    )

    dropout_layer = tf.keras.layers.Dropout(.2)

    dense_layer_output = tf.keras.layers.Dense(
        units = 4,
        bias_initializer = tf.keras.initializers.glorot_normal(),
```

```
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        activation = tf.keras.activations.softmax
    )

    model.add(input_conv_layer)
    model.add(conv_layer_2)
    model.add(conv2_flat)
    model.add(dense_layer_1)
    model.add(dropout_layer)
    model.add(dense_layer_2)
    model.add(dense_layer_3)
    model.add(dropout_layer)
    model.add(dense_layer_4)
    model.add(dropout_layer)
    model.add(dense_layer_output)
    model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = lr), loss=tf.keras.l
    return model
```

Modellentwurf 4

Da 4te und letzte Modell versucht, die im 3ten Modell verwendeten Ansätze weiter zu verbessern. So wird die Anzahl der Faltungsschichten wieder auf 3 erhöht:

1. Das Layout der ersten Faltung bleibt identisch
2. Die zweite Faltungsschicht ist eine verkleinere Version der ersten Schicht. Statt Über 10 Frames wird nur noch über 5 Frames gefaltet.
3. Die 3te Schicht ist identisch zur 2ten Schicht des 3ten Modells

Das fully connected Netz wird im Vergleich zum 3ten Model um eine Schicht reduziert. Alles in allem wird die Parameteranzahl im Vergleich fast nochmal halbiert.

Netz Zusammenfassung:

Layer (type)	Output Shape	Param #
=====		
conv2d_37 (Conv2D)	(None, 81, 25, 32)	992
conv2d_38 (Conv2D)	(None, 39, 25, 16)	2576
conv2d_39 (Conv2D)	(None, 18, 1, 32)	64032
flatten_15 (Flatten)	(None, 576)	0
dense_68 (Dense)	(None, 100)	57700
dropout_23 (Dropout)	multiple	0
dense_69 (Dense)	(None, 50)	5050
dense_70 (Dense)	(None, 20)	1020
dense_71 (Dense)	(None, 4)	84
=====		
Total params: 131,454		
Trainable params: 131,454		
Non-trainable params: 0		

In []:

```
def define_and_compile_fourth_model(lr=0.001):
    model = tf.keras.Sequential()
    input_conv_layer = tf.keras.layers.Conv2D(
        input_shape = (170,25,3),
        data_format = 'channels_last',
        kernel_size = (10,1),
        strides = (2,1),
        padding = 'valid',
        activation = tf.keras.activations.relu,
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        filters = 32
    )

    conv_layer_2 = tf.keras.layers.Conv2D(
        data_format = 'channels_last',
        kernel_size = (5,1),
        strides = (2,1),
        padding = 'valid',
        activation = tf.keras.activations.relu,
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        filters = 16
    )

    conv_layer_3 = tf.keras.layers.Conv2D(
        data_format = 'channels_last',
        kernel_size = (5,25),
        strides = 2,
        padding = 'valid',
        activation = tf.keras.activations.relu,
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        filters = 32
    )

    conv2_flat = tf.keras.layers.Flatten()

    dense_layer_1 = tf.keras.layers.Dense(
        activation = tf.keras.activations.relu,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        units = 100
    )

    dense_layer_2 = tf.keras.layers.Dense(
        activation = tf.keras.activations.relu,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        units = 50
    )

    dense_layer_3 = tf.keras.layers.Dense(
        activation = tf.keras.activations.relu,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        units = 20
    )

    dropout_layer = tf.keras.layers.Dropout(.2)

    dense_layer_output = tf.keras.layers.Dense(
```

```

        units = 4,
        bias_initializer = tf.keras.initializers.glorot_normal(),
        kernel_initializer = tf.keras.initializers.glorot_normal(),
        activation = tf.keras.activations.softmax
    )

model.add(input_conv_layer)
model.add(conv_layer_2)
model.add(conv_layer_3)
model.add(conv2_flat)
model.add(dense_layer_1)
model.add(dropout_layer)
model.add(dense_layer_2)
model.add(dropout_layer)
model.add(dense_layer_3)
model.add(dropout_layer)
model.add(dense_layer_output)
model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = lr), loss=tf.keras.l
return model

```

Methoden zur Evaluierung

Für die Evaluierung der Netze wurde sich auf folgende Metriken festgelegt:

1. Das Training der Modelle zielt auch die Accuracy Metrik. Bei unbalancierten Datensätzen wäre die Accuracy skeptischer zu betrachten. Da unsere Klassen gleich oft vertreten sind, stellt diese Metrikt keine Probleme dar.
2. Die Konfusionsmatrix visualisiert ganz gut, mit welchen Aktionsklassen unsere Modelle Probleme haben. sie fasst gleichzeitig die Precision und den Recall für alle Klassen an.
3. Die ROC curve, oder auch Receiver operating characteristic curve. diese Vergleicht für jede Klasse den Recall sowie die Sensitivity. Da wir nicht mit einem Binären Klassifikator arbeiten, verwenden wir für die ROC nur die Area Under Curve. Berechnet wird diese im Multiclass Fall durch den Mittelwert. Je näher dieser an 1 liegt, desto besser. Auch hier gilt: Da unsere Klassen balanciert sind, stellt der Mittelwert kein Problem dar.
4. Der F-1 Score. Für jede Klasse wird zusätzlich noch aus Precision und Recall der F-1 Score berechnet. Dies dient zusätzlich zur Konfusions Matrix dazu, die Vorhersagegenauigkeit des Modells bei einzelnen Klassen besser bewerten zu können. Der Mittelwert des F-1 Score stellt im übrigen eine gute Metrik für die Bewertung des Modells dar.

Berechnet werden die Metriken sowohl auf Test als auch auf Trainingsdaten. so lässt sich ein Overfitting besser feststellen. Auch kann es sein, dass nicht alle Trainingsdaten am ende in unser Modell mit eingeflossen sind. Dies ist insbesondere dann der Fall, wenn wir das Training frühzeitig abbrechen, da die Genauigkeit und/oder der Loss wieder abnehmen.

In []:

```

def evaluate_and_save(model, model_name, train, test, save = False):
    print('Evaluating with training Dataset')
    eval_train = model.evaluate(train[0], train[1], batch_size = 10, verbose = 1)
    pred_train = model.predict(train[0], batch_size = 10, verbose = 1)
    pred_train_class = np.argmax(pred_train, axis = 1)
    pred_train_onehot = to_categorical(pred_train_class)
    pred_train_acc = accuracy_score(train[1], pred_train_onehot)
    pred_train_conf_mat = confusion_matrix(np.argmax(train[1], axis = 1), pred_train_class)
    print(pred_train_acc)
    print(pred_train_conf_mat)
    pred = model.predict(X_test, batch_size = 10, verbose = 1)
    print(classification_report(train[1], pred_train_onehot, target_names=target_names))
    print('ROC Area under Curve Score: ' + str(roc_auc_score(train[1], pred_train_onehot, lab

print('_____

eval_test = model.evaluate(test[0], test[1], batch_size = 10, verbose = 1)
pred_test = model.predict(test[0], batch_size = 10, verbose = 1)
pred_test_class = np.argmax(pred_test, axis = 1)
pred_test_onehot = to_categorical(pred_test_class)
pred_test_acc = accuracy_score(test[1], pred_test_onehot)
pred_test_conf_mat = confusion_matrix(np.argmax(test[1], axis = 1), pred_test_class)
print(pred_test_acc)
print(pred_test_conf_mat)

print(classification_report(test[1], pred_test_onehot, target_names=target_names))
print('ROC Area under Curve Score: ' + str(roc_auc_score(test[1], pred_test_onehot, label
timestr = time.strftime("%Y%m%d-%H%M%S"))
filename = '/content/drive/My Drive/Studium/Projektarbeit/Models/' + model_name + '_' + tim

print('_____
print('_____

if save:
    model.save(filename+'.h5')
    f= open(filename+'.txt', "w+")
    text = 'train accuracy: ' + str(pred_train_acc) + '\n'
    text += str(pred_train_conf_mat) + '\n'
    text += classification_report(train[1], pred_train_onehot, target_names=target_names) +
    text += 'ROC Area under Curve Score: ' + str(roc_auc_score(train[1], pred_train_onehot,
    text += '----- \n'
    text += 'test accuracy: ' + str(pred_test_acc) + '\n'
    text += str(pred_test_conf_mat) + '\n'
    text += classification_report(test[1], pred_test_onehot, target_names=target_names) +
    text += 'ROC Area under Curve Score: ' + str(roc_auc_score(test[1], pred_test_onehot, la

    f.write(text+'\n')
    f.close()

```

Modelltraining

Beim Trainieren der Modelle hinterlegen wir einen Checkpoint, sowie einen TensorBoard Callback. Der Tensorboard callback erlaubt es uns, die Metriken des Trainings abzuspeichern und in Tensorboard nachträglich zu untersuchen. Der checkpoint Callback wiederum speichert nach jeder Epoche die Gewichte des

Modells ab, sofern das Modell bessere Validierungs-Ergebnisse liefert. Am Ende des Durchlaufs können wir dann auf die Gewichte zurückgreifen, welche die besten Resultate erbracht haben.

In []:

```
def fit_and_eval_model(model, model_name, X_train, y_train, X_test, y_test):  
    logdir = '/content/drive/My Drive/Studium/Projektarbeit/logs/' + model_name  
  
    tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)  
  
    model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(  
        filepath='/tmp/checkpoints',  
        save_weights_only=True,  
        monitor='val_accuracy',  
        mode='max',  
        save_best_only=True)  
  
    model.fit(X_train, y_train, epochs = 60, batch_size=64, validation_data=(X_test, y_test))  
    model.load_weights('/tmp/checkpoints')  
    evaluate_and_save(model, model_name, [X_train, y_train], [X_test, y_test], True)
```

Für das Training bestimmte Daten müssen durch das auskommentieren der entsprechenden Codezeilen ausgewählt werden

In []:

```

#Für Unnormalisierte Daten:
#X_train, X_test, y_train, y_test, name_prefix = preparation_pipeline(None)

#Für normalisierte Daten nach Körperkoordinaten pro Frame:
#X_train, X_test, y_train, y_test, name_prefix = preparation_pipeline('normal')

#Für normalisierte Daten nach Körperkoordinaten nach erstem Frame:
#X_train, X_test, y_train, y_test, name_prefix = preparation_pipeline('advanced')

#Für rotationsnormalisierte Daten nach Körperkoordinaten:
X_train, X_test, y_train, y_test, name_prefix = preparation_pipeline('rotation')

#Für Cross View Daten:
#X_train, X_test, y_train, y_test = cross_view_train_X, cross_view_test_X, cross_view_train
#name_prefix = 'cross_view_'

#Für Cross Subject Daten:
#X_train, X_test, y_train, y_test = cross_subject_train_X, cross_subject_test_X, cross_subj
#name_prefix = 'cross_subject_'

learning_rate = 0.0005
models = [define_and_compile_model(learning_rate), define_and_compile_second_model(learning
for index, model in enumerate(models,1):
    name = name_prefix + 'model_regularized_' + str(index)
    print(model.summary())
    fit_and_eval_model(model, name, X_train, y_train, X_test, y_test)

```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 166, 25, 32)	512
max_pooling2d (MaxPooling2D)	(None, 162, 25, 32)	0
conv2d_1 (Conv2D)	(None, 158, 25, 32)	5152
max_pooling2d_1 (MaxPooling2	(None, 154, 25, 32)	0
flatten (Flatten)	(None, 123200)	0
dense (Dense)	(None, 100)	12320100
dropout (Dropout)	(None, 100)	0
dense_1 (Dense)	(None, 50)	5050

Resultate

Eine Grobe Übersicht über die trainierten Netze liefert Tensorboard. Hier können wir die Logs aller unserer trainierten Modelle einsehen. Eine vertiefte Einsicht bieten darüber hinaus die Metriken unserer Evaluation. Nachdem wir so unser bestes Modell ausgewählt haben, können wir es mit dem alternativen Ansatz

vergleichen, bei dem RNN statt CNN genutzt wurden.

Analyse in Tensorboard

Tensorboard erlaubt uns, wie bereits erwähnt, die Logs aller unserer Modelle einzusehen. Zusätzlich können wir die Logs der zum Vergleich trainierten RNN ebenfalls laden und so den Trainingsverlauf beider Netze direkt vergleichen.

Ein Blick auf den Validierungs-Loss

Beobachtet man den Verlauf des Validierungs-Loss beim Trainieren des Modells, so stellt man fest, dass dieser ab einem bestimmten Zeitpunkt wieder stark ansteigt. Im Normalfall ist dies ein klares Indiz für ein Overfitting. Allerdings bringt ein solches Overfitting generell einen Verlust an Validierungsgenauigkeit mit sich. Dies ist hier jedoch nicht der Fall. Im Gegenteil, die Genauigkeit steigt sogar teilweise noch an oder bleibt schlimmstenfalls stabil.

Woran liegt dies?

Der Output unseres Modells ist ein Softmax. Das bedeutet, dass wir 4 Ausgangswerte haben, welche normiert und aufsummiert den Wert 1 ergeben. Somit können die Outputwerte als Wahrscheinlichkeitswert interpretiert werden, mit dem das Modell seine Vorhersage trifft. In unserem Fall ist es nun so, dass das Modell im Verlauf des Trainings öfters richtige Vorhersagen trifft, diese Wahrscheinlichkeitswerte für die richtige Antwort allerdings sinken. Das Modell wird sich in seiner Antwort also "weniger sicher", obwohl die Genauigkeit steigt. Eine schöne Erklärung findet sich auch in diesem [Stackexchange Beitrag](https://stats.stackexchange.com/questions/282160/how-is-it-possible-that-validation-loss-is-increasing-while-validation-accuracy). (<https://stats.stackexchange.com/questions/282160/how-is-it-possible-that-validation-loss-is-increasing-while-validation-accuracy>).

Eine Möglichkeit, diesem Effekt entgegenzuwirken ist die Regularisierung. Durch das Einfügen von Dropout-Layern konnte der Effekt gedämpft und die Performance des Modells nochmal verbessert werden. Allerdings konnte der Anstieg des Loss nie komplett unterbunden werden. Dies ist im Tensorboard gut sichtbar.

Vergleich der genutzten Daten

Vergleicht man die Trainingsverläufe der einzelnen Modell-Architekturen mit den unterschiedlich normalisierten Daten, so zeigt sich ganz klar ein Muster ab. Eine stärkere Normalisierung führt zu:

- einem schnelleren Lernen des Netzes
- einer höheren Genauigkeit für Validierungs- und Trainingsdaten
- einem geringeren Wiederanstieg des Validierungs-Loss

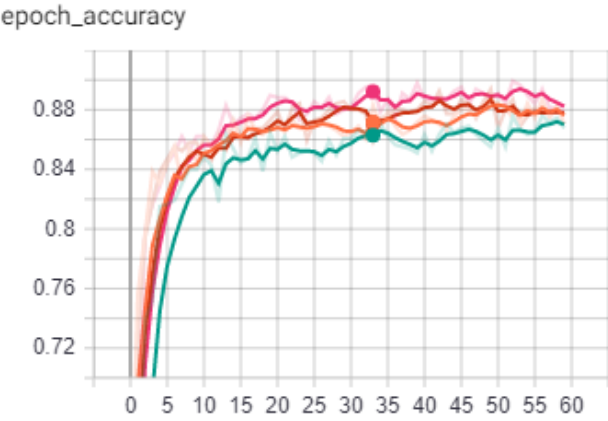
Somit wird deutlich, wie wichtig eine gute Normalisierung der Trainingsdaten ist und welchen weitreichenden Einfluss sie auf das Ergebnis des Modells hat.

Vergleich der CNN untereinander

Vergleichen wir die verschiedenen CNN untereinander, so wird schnell deutlich, dass Modell 4 was Performance angeht die Nase vorn hat. Die genauen Metrikerwerte sind im nächsten Abschnitt einsehbar.

Vergleich der Verschiedenen Modellarchitekturen

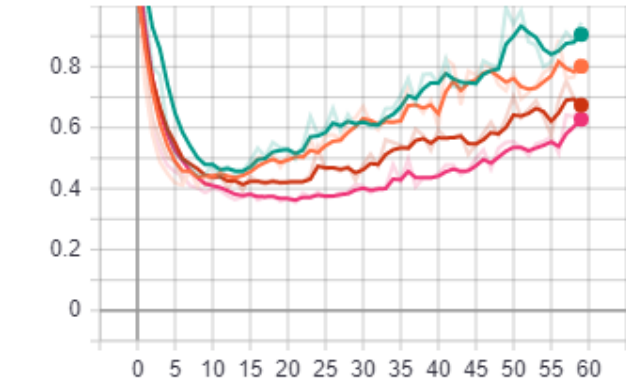
Validierungs Genauigkeit



	Name	Smoothed	Value	Step	Time	Relative
●	rotation_normalized_model_regularized_1/validation	0.8629	0.8634	33	Wed Sep 23, 13:38:22	2m 2s
●	rotation_normalized_model_regularized_2/validation	0.8717	0.8828	33	Wed Sep 23, 13:42:24	2m 14s
●	rotation_normalized_model_regularized_3/validation	0.8713	0.8593	33	Wed Sep 23, 13:44:48	27s
●	rotation_normalized_model_regularized_4/validation	0.8921	0.8966	33	Wed Sep 23, 13:45:45	28s

Validierungs Loss

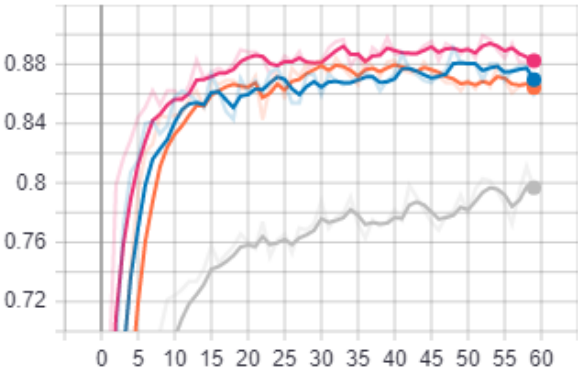
	Name	Smoothed	Value	Step	Time	Relative
●	rotation_normalized_model_regularized_1/validation	0.9062	0.9454	59	Wed Sep 23, 13:39:57	3m 36s
●	rotation_normalized_model_regularized_2/validation	0.8014	0.828	59	Wed Sep 23, 13:44:10	4m 1s
●	rotation_normalized_model_regularized_3/validation	0.6743	0.6451	59	Wed Sep 23, 13:45:10	49s
●	rotation_normalized_model_regularized_4/validation	0.6278	0.6654	59	Wed Sep 23, 13:46:07	50s







Vergleicht der Normalisierungen anhand von Modell 4





Validierungs Genauigkeit

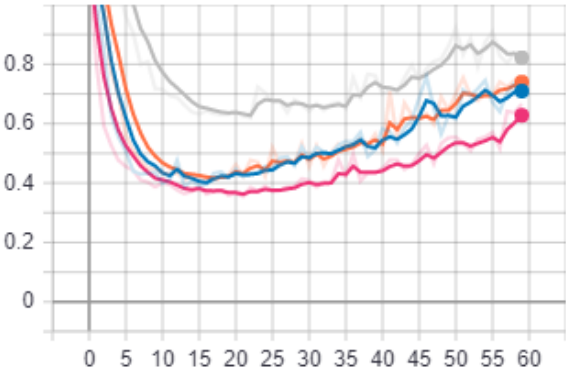
epoch_accuracy



Name	Smoothed	Value	Step	Time	Relative
 advanced_normalized_model_regularized_4/validation	0.8696	0.8579	59	Wed Sep 23, 11:18:45	50s
 normalized_model_regularized_4/validation	0.8642	0.8607	59	Wed Sep 23, 10:36:18	50s
 rotation_normalized_model_regularized_4/validation	0.8823	0.8786	59	Wed Sep 23, 13:46:07	50s
 unnormalized_model_regularized_4/validation	0.7968	0.7959	59	Wed Sep 23, 10:23:00	49s

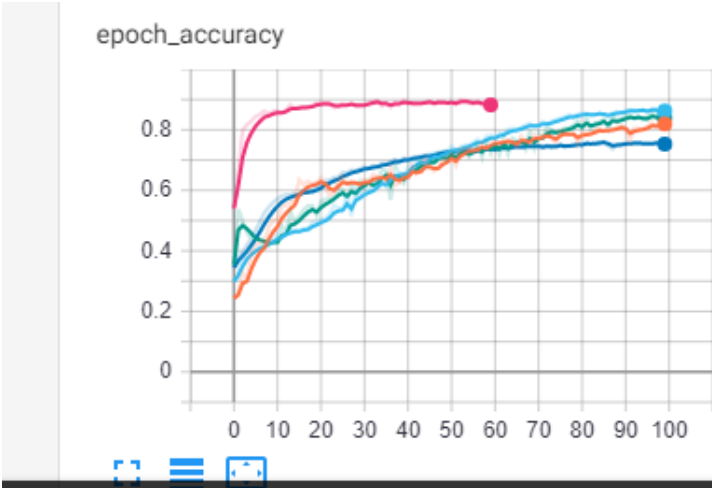
Validierungs Loss

Name	Smoothed	Value	Step	Time	Relative
 advanced_normalized_model_regularized_4/validation	0.7095	0.7147	59	Wed Sep 23, 11:18:45	50s
 normalized_model_regularized_4/validation	0.7398	0.7568	59	Wed Sep 23, 10:36:18	50s
 rotation_normalized_model_regularized_4/validation	0.6278	0.6654	59	Wed Sep 23, 13:46:07	50s
 unnormalized_model_regularized_4/validation	0.8215	0.8004	59	Wed Sep 23, 10:23:00	49s



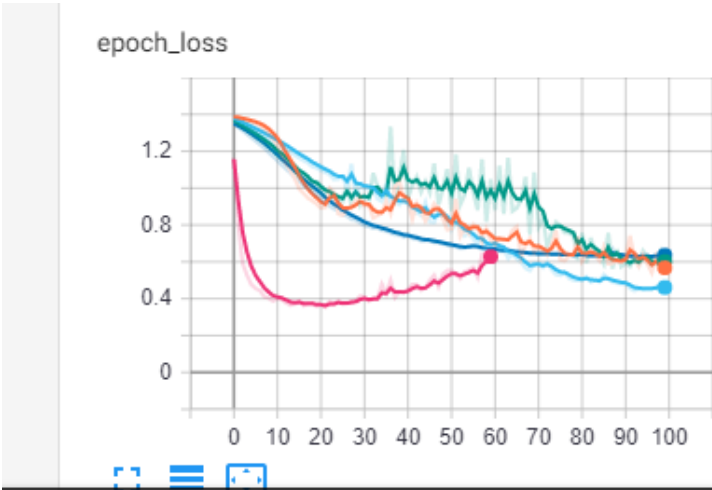
Vergleicht zwischen RNN und CNN

Validierungs Genauigkeit



Name	Smoothed	Value	Step
RNN_Models/RNN_NTU_fillup_3D_classes_4_layers_2_cc_128_1599572057/validation	0.8195	0.8273	99
RNN_Models/RNN_NTU_fillup_norm_3D_classes_4_layers_2_cc_128_1599481561/validation	0.8622	0.8715	99
RNN_Models/RNN_NTU_fillup_norm_3D_classes_4_layers_2_cc_160_1599564425/validation	0.8395	0.8412	99
RNN_Models/RNN_NTU_sliding_norm_3D_classes_4_layers_2_cc_64_1599465002/validation	0.7529	0.7511	99
rotation_normalized_model_regularized_4/validation	0.8823	0.8786	59

Validierungs Loss



Name	Smoothed	Value	Step
RNN_Models/RNN_NTU_fillup_3D_classes_4_layers_2_cc_128_1599572057/validation	0.5678	0.5431	99
RNN_Models/RNN_NTU_fillup_norm_3D_classes_4_layers_2_cc_128_1599481561/validation	0.461	0.4531	99
RNN_Models/RNN_NTU_fillup_norm_3D_classes_4_layers_2_cc_160_1599564425/validation	0.6021	0.5741	99
RNN_Models/RNN_NTU_sliding_norm_3D_classes_4_layers_2_cc_64_1599465002/validation	0.6357	0.6339	99
rotation_normalized_model_regularized_4/validation	0.6278	0.6654	59

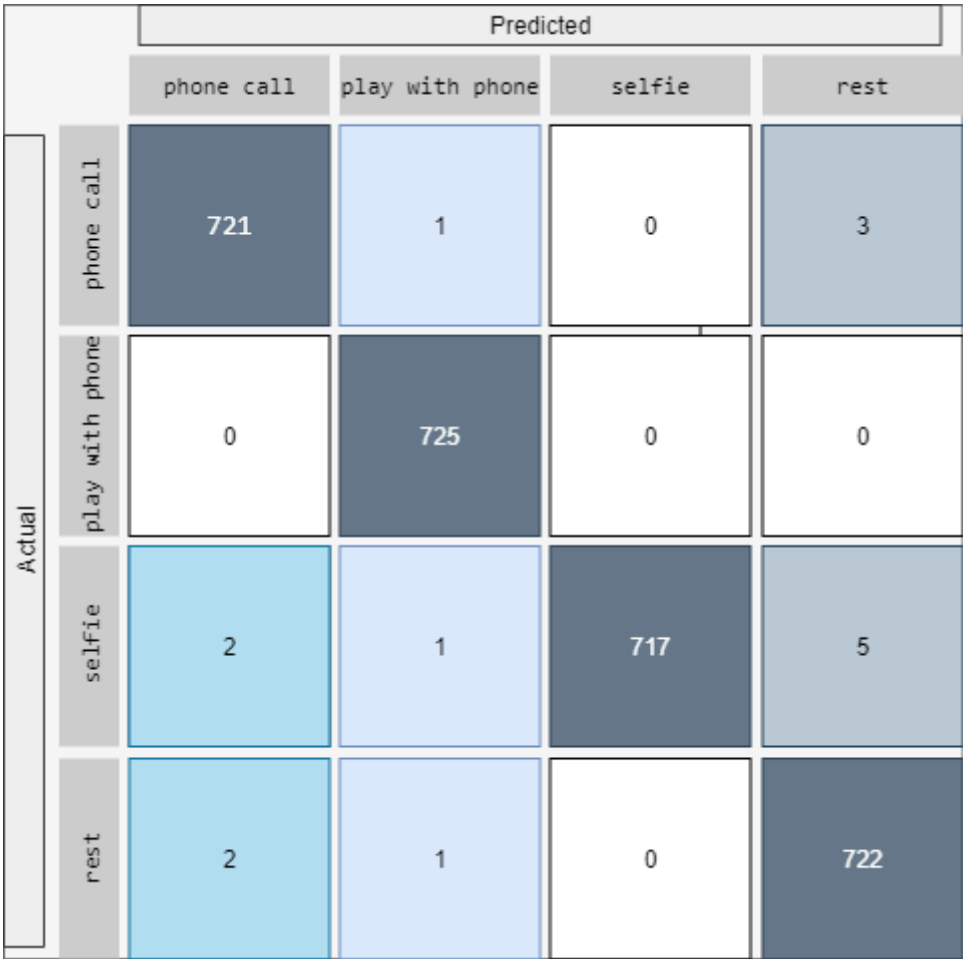
Metrikwerte

Für die ausgewählten Metriken erreicht das Modell 4 folgende Werte:

Auf Trainingsdaten:

Accuracy: 0.9948

Konfusions Matrix:



Precision, Recall und F1-score für die Klassen:

class	precision	recall	f1-score	support
phone call	0.99	0.99	0.99	725
play with phone	1.00	1.00	1.00	725
taking a selfie	1.00	0.99	0.99	725
rest class	0.99	1.00	0.99	725

Precision Recall und F1-score im Schnitt:

Average Type	precision	recall	f1-score	support
micro avg	0.99	0.99	0.99	2900
macro avg	0.99	0.99	0.99	2900
weighted avg	0.99	0.99	0.99	2900
samples avg	0.99	0.99	0.99	2900

ROC Area under Curve Score: 0.9966

Auf Testdaten:

Accuracy: 0.8993

Konfusions Matrix:

		Predicted			
		phone call	play with phone	taking a selfie	rest class
Actual	phone call	159	6	3	14
	play with phone	8	168	1	4
	taking a selfie	6	1	160	14
	rest class	4	6	6	165

Precision, Recall und F1-score für die Klassen:

class	precision	recall	f1-score	support
phone call	0.90	0.87	0.89	182
play with phone	0.93	0.93	0.93	181
taking a selfie	0.94	0.88	0.91	181
rest class	0.84	0.91	0.87	181

Precision Recall und F1-score im Schnitt:

Average Type	precision	recall	f1-score	support
micro avg	0.90	0.90	0.90	725
macro avg	0.90	0.90	0.90	725
weighted avg	0.90	0.90	0.90	725
samples avg	0.90	0.90	0.90	725

ROC Area under Curve Score: 0.9329

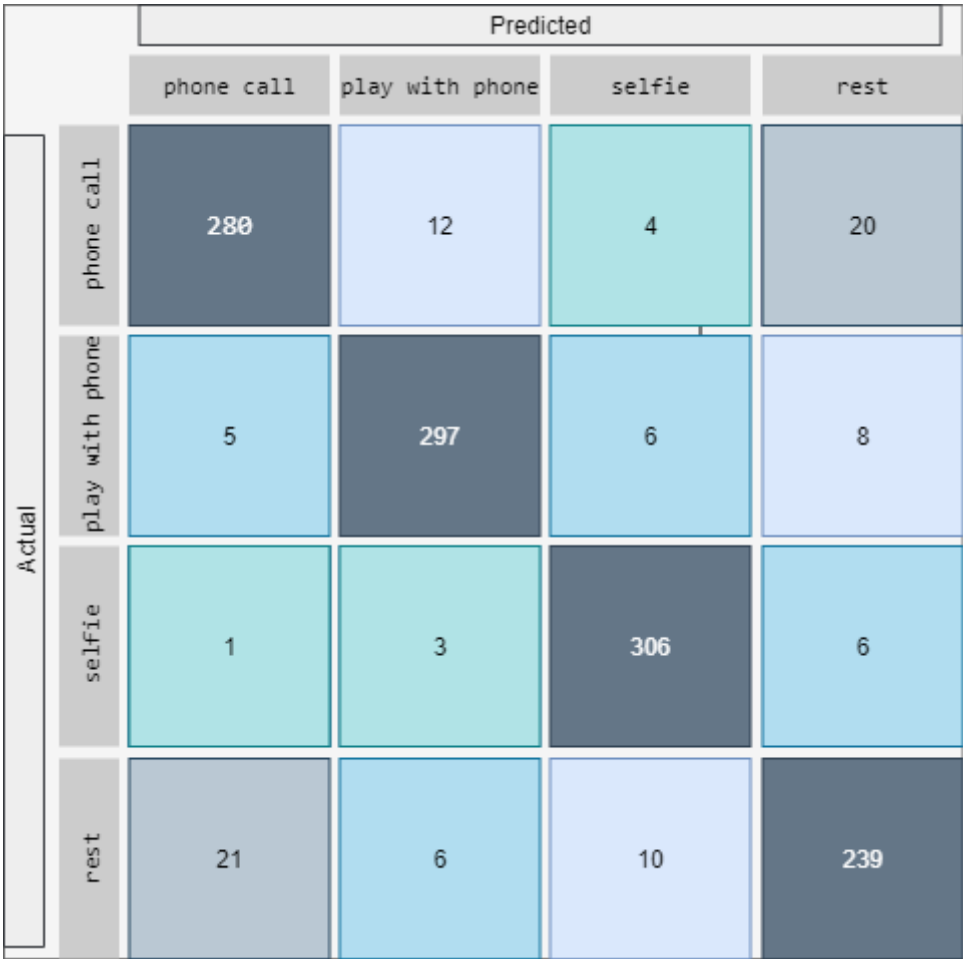
Cross View und Cross Subject Auswertung

Ebenfalls mit Modell 4

Für Cross View:

Accuracy: 0.9167

Konfusions Matrix:



Precision, Recall und F1-score für die Klassen:

class	precision	recall	f1-score	support
phone call	0.91	0.89	0.90	316
play with phone	0.93	0.94	0.94	316
taking a selfie	0.94	0.97	0.95	316
rest class	0.88	0.87	0.87	276

Precision Recall und F1-score im Schnitt:

Average Type	precision	recall	f1-score	support
micro avg	0.92	0.92	0.92	1224
macro avg	0.92	0.92	0.91	1224
weighted avg	0.92	0.92	0.92	1224
samples avg	0.92	0.92	0.92	1224

ROC Area under Curve Score: 0.9437

Für Cross Subject:

Accuracy: 0.8811

Konfusions Matrix:

		Predicted			
		phone call	play with phone	selfie	rest
Actual	phone call	230	15	8	22
	play with phone	7	256	5	7
	selfie	5	4	257	10
	rest	25	13	10	228

Precision, Recall und F1-score für die Klassen:

class	precision	recall	f1-score	support
phone call	0.86	0.84	0.85	275
play with phone	0.89	0.93	0.91	275
taking a selfie	0.92	0.93	0.92	276
rest class	0.85	0.83	0.84	276

Precision Recall und F1-score im Schnitt:

Average Type	precision	recall	f1-score	support
micro avg	0.88	0.88	0.88	1102
macro avg	0.88	0.88	0.88	1102
weighted avg	0.88	0.88	0.88	1102
samples avg	0.88	0.88	0.88	1102

ROC Area under Curve Score: 0.9208

Interpretation der Metriken

Im normalen Train/Test Split

die Konfusionsmatrix zeigt deutlich, dass die Restklasse die größte Herausforderung darstellt. Dies stellt insofern keine Überraschung dar, da die 3 Hauptaktionen recht unterschiedlicher Natur sind, die Restklasse hingegen Bewegungsabläufe enthält, die denen der Aktionen teilweise sehr ähnlich sind.

So sind "Hut absetzen" und "Wasser trinken" beispielsweise beide Aktionen, welche eine Bewegung der Hand zum Kopf hin beinhalten. Die gleiche oder eine ähnliche Bewegung findet sich jedoch auch bei der Aktion "phonecall" oder "taking a selfie".

"Klatschen" mit kurzen Handbewegungen ist auf Skelettebene wiederum leicht mit "Play with Phone" zu verwechseln.

Das Faltungsnetz ordnet somit teilweise Sequenzen fälschlicherweise der Restklasse zu.

Überraschen ist allerdings, wie oft "phonecall" und "play with phone" verwechselt werden. Dies ist auf die Tatsache zurückzuführen, dass vor der Aktion des Anrufens oft die vorher die Telefonnummer eingetippt wird. Diese scheinbar harmlose Information führt dazu, dass das Modell darin eine "play with phone" Aktion erkennt.

Cross View und Cross Subject

Hier zeigt sich eindeutig, dass das Problem der Cross View Klassifizierung bei weitem eine leichtere Aufgabe darstellt. Das Netz liefert jedoch bei beiden Problemen eine zufriedenstellende Leistung. Anzumerken ist ebenfalls die Tatsache, dass eine gezielte Betrachtung des Cross Subject Problems eine niedrigere Vorhersagegenauigkeit ergibt als bei dem normalen Train/Test Split. Das Hauptproblem stellt jedoch bei beiden Ansätzen weiterhin die Restklasse dar. Die Unterscheidung zwischen den 3 Hauptaktionen "play with phone", "take selfie" und "phonecall" ist weiterhin sehr zufriedenstellend. Die vorher festgestellten Verwechslungspotentiale bei den Aktionsklassen werden insbesondere bei Cross-Subject jedoch deutlicher.

Vergleich zu Rekurrenten Neuronalen Netzen

Übersicht der RNN

Auf die genaue Architektur des Rekurrenten LSTM Netzes werden wir in dieser Ausarbeitung nicht im Detail eingehen. Hierfür empfiehlt sich die Lektüre der Arbeit von Miro Goettler, welcher jenen Teilbereich des Projektes übernommen hat.

Mit einer Parameteranzahl von 237,060 ist das Rekurrente Netz jedoch in einer Größenkategorie mit den entgeltigen vorgestellten Faltungsnetzen.

Die Analyse der Logs in Tensorboard

Der Trainingsverlauf der RNN zeigt einen stetigeren, wenngleich auch langsameren Anstieg der Vorhersagegenauigkeit. Auch verzeichnet das Modell keinen Wiederanstieg beim Validierungs-Loss. Im Gegensatz zu den Faltungsnetzen scheinen die RNN zu Trainingsende noch kein Leistungs Plateau zu erreichen. Es könnte davon ausgegangen werden, dass eine höheren Anzahl an Trainingsdaten die Genauigkeit noch weiter ansteigen lassen würde.

Die Metriken der RNN

Die Metrikwerte für das am besten abschneidende RNN sehen wie folgt aus:

Parameteranzahl: 237,060 **Accuracy:** 0.8715

Konfusions Matrix:

		Predicted			
		phone call	play with phone	taking a selfie	rest class
Actual	phone call	154	15	2	10
	play with phone	9	169	2	1
	taking a selfie	9	3	163	6
	rest class	15	14	7	145

Precision, Recall und F1-score für die Klassen:

class	precision	recall	f1-score	support
phone call	0.82	0.85	0.84	181
play with phone	0.84	0.93	0.88	181
taking a selfie	0.94	0.90	0.92	181
rest class	0.90	0.80	0.85	181

Precision Recall und F1-score im Schnitt:

Average Type	precision	recall	f1-score	support
macro avg	0.87	0.87	0.87	724
weighted avg	0.87	0.87	0.87	724

ROC Area under Curve Score: 0.9723

Im Vergleich zur Konfusionsmatrix der Faltungsnetze springt ein Unterschied ins Auge. Während das Faltungsnetz öfters andere Aktionen fälschlicherweise der Restklasse zuordnet, verläuft dies beim RNN genau umgekehrt. So werden Sequenzen der Restklasse nicht als solche erkannt und eher den anderen Klassen zugewiesen, hauptsächlich den "phonecall" und "play with phone" Klassen.

Für Cross View und Cross Subject:

Cross View:

Accuracy: 0.8897

Cross Subject:

Accuracy: 0.8348

Vergleich der Metriken der Modelle**Parameteranzahl:**

RNN: 237,060

CNN: 131,454

Accuracy:

CNN: 0.8993

RNN: 0.8715

class	precision RNN	precision CNN	recall RNN	recall CNN	f1-score RNN	f1-score CNN	support
phone call	0.82	0.90	0.85	0.87	0.84	0.89	181
play with phone	0.84	0.93	0.93	0.93	0.88	0.93	181
taking a selfie	0.94	0.94	0.90	0.88	0.92	0.91	181
rest class	0.90	0.84	0.80	0.91	0.85	0.87	181

Precision Recall und F1-score im Schnitt:

Average Type	precision RNN	precision CNN	recall RNN	recall CNN	f1-score RNN	f1-score CNN	support
macro avg	0.87	0.90	0.87	0.90	0.87	0.90	724
weighted avg	0.87	0.90	0.87	0.90	0.87	0.90	724

ROC Area under Curve Score:

CNN: 0.9329

RNN: 0.9723

Die vorherigen Festellungen scheinen sich im direkten Vergleich der Precision und Recall Werte zu bestätigen:

- Die Precision der RNN ist geringer für "phonecall" und "play with phone". Da die falsch zugeordneten Sequenzen der Restklasse entstammen ist der Recall Wert für die Restklasse entsprechend gering.
- Das Faltungsnetz versucht öfters, Sequenzen der Restklasse zuzuordnen, somit ist die Precision für die Restklasse gering, der Recall jedoch sehr hoch.

Im Mittel über die Klassen hinweg schneidet das Faltungsnetz bei Precision, Recall und somit auch dem F1-Score jedoch besser ab.

Beachtlich ist jedoch die Area Under Curve für die ROC! Im Multiklassen Verfahren werden für die Berechnung der ROC jeweils zwei Klassen miteinander verglichen und dann ein Mittelwert berechnet. So besagt ein hoher ROC in diesem Fall, dass das RNN prinzipiell besser zwischen 2 Aktionen unterscheiden kann als das CNN.

Cross View Accuracy

CNN: 0.9167

RNN: 0.8897

Cross Subject Accuracy

CNN: 0.8811

RNN: 0.8348

Die Precision Werte für Cross View und Cross Subject bestätigen eigentlich nur die vorherigen Metriken.

Fazit

Sowohl CNN als auch RNN eignen sich zur Erkennung von Bewegungsabläufen auf Skelettdaten. Einen enormen Vorteil bringt hierbei das korrekte Normalisieren der verwendeten Daten. Unterschiede der beiden Netzstrukturen lassen sich beim Training der Netze beobachten. Hier trainieren die Faltungsnetze deutlich schneller! Allerdings scheint das Potential der Rekurrenten Netze nicht ganz ausgenutzt, da am Ende des Trainings im Gegensatz zu den CNN kein Leistungsplateau erreicht wurde und kein Overfitting zu erkennen ist. Größte Schwachstelle der RNN scheint die Restklasse zu sein, welche sich aus gleichen Teilen anderer Bewegungsabläufe zusammensetzt. Hier ist die Eigenschaft der CNN, sehr schnell zu lernen, von Vorteil. Da die verschiedenen Bewegungen innerhalb Restklasse vergleichsweise wenig oft vertreten sind, kann das RNN diese nicht komplett erlernen und die Restklasse infolgedessen nur bedingt generalisieren.

Erwähnenswert ist jedoch die Tatsache, dass das Faltungsnetz mit knapp der Hälfte an trainierbaren Parametern auskommt und das Training für eine bessere Performance sogar vorzeitig abgebrochen wird. Das Faltungsnetz nutzt somit nur einen Teil der potentiellen Trainingsdaten. Eine große Rolle bei der Lernrate des Netzes spielt höchswahrscheinlich auch die Anpassbarkeit der Faltungskerne. Vergleicht man die Resultate der 4 vorgestellten möglichen Architekturen so zeigt sich, wie viel die Form und Reihenfolge der Faltungskerne ausmachen kann.

In []: