h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbmn
FACHBEREICH MATHEMATIK
UND NATURWISSENSCHAFTEN

Projekt 1:

Handgestenerkennung mit einer Smart-Cam

unter Verwendung eines neuronalen Netzes (Kurzanleitung)

Team:
Franka Ludig, Jan Vincent Sievers,
Julian Odloschinski und Lukas Reißer

Betreuer:
Herr Prof. Neubecker

Einführung

Wir haben in unserem Projekt 1 ein neuronales Netz in ein Programm implementiert, welches auf einer uns zur Verfügung gestellten Smart Camera (Vision Sensor PV) läuft und 10 unterschiedliche Handgesten erkennen kann. Eine Smart Kamera ist kurz gesagt eine Kamera, welche einen Prozessor integriert hat und somit in der Laufzeit keinen externen Computer, Bildschirm o.ä benötigt. Die Kamera läuft mit einem Debian Betriebssystem. Mit Hilfe einer SSH Verbindung kann auf die Kamera zugegriffen werden, dazu später mehr. Ein künstliches neuronales Netzwerk ist an einem biologischen neuronalen Netzwerk (Gehirn) angelehnt, welches künstliche Neuronen/Verbindungen zur Informationsverarbeitung nutzt. Kurz gesagt kann man ein neuronales Netzwerk (NN) mit Testbildern von z.B. Handgesten so trainieren, dass das NN diese Handgesten erkennt. Für unser NN wurden 20 000 unterschiedliche Bilder zum Trainieren benutzt. Mehr Informationen zum NN gibt es hier vom Entwickler.

Vorgaben des Projektes:

Wir haben von der Firma IMAGO die Smart Cam Vision Sensor PV zur Verfügung gestellt bekommen. Unser Ziel ist es ein neuronales Netz auf der Kamera einzurichten und das Ergebnis ggf. auf einem LED-Feld ausgeben zu lassen.

Das neuronale Netz mit einer kurzen Beschreibung gibt es hier.

Die geplante Vorgehensweise:

- Zugriff zur Kamera herstellen (SSH)
- Testprogramm auf der Kamera laufen lassen
- NN zum laufen bringen
- Das in Python geschriebene NN mit dem Kamera Code (C++) verknüpfen und zum laufen zu bringen
- Programm automatisieren/verbessern

Erste Schritte:

In unserem Fall war die Kamera schon so eingerichtet, dass wir uns nur noch mit dieser verbinden mussten. Falls das noch nicht geschehen ist, hat Imago zum Einrichten der Kamera ein Tutorial auf YouTube hochgeladen, welches auch das Verbinden von MobaXTerm mit der Kamera beinhaltet.

Der Zugriff zur Kamera per MobaXTerm bietet eine Ordnerübersicht welche die generelle Übersicht stark verbessert und die Funktion bereitstellt Dateien per Drag&Drop auf den eigenen Rechner zu ziehen.

Bei uns gab es allerdings zeitweise Probleme mit MobaXTerm, weshalb wir hier noch ein kurzes Tutorial für die ersten Schritte mit PowerShell geschrieben haben.

Hinweis: Der Zugriff auf die Kamera muss nicht als Root-User erfolgen, da die benötigten Kommandos auch mit *sudo* ausgeführt werden können. Aus Sicherheitsgründen wird hier darauf verzichtet, sich als Root-User anzumelden.

Um eine Verbindung zur Kamera herstellen zu können, muss man sich mit SSH über ein Terminal (bspw. Power Shell) auf der Kamera anmelden. Mit folgendem Befehl ist das möglich:

ssh visionsensor@LOKALE_IP_ADRESSE

```
PS C:\Users\Nutzer> ssh visionsensor@141.100.188.72
>>
>> %%  Die Ip Adresse weicht möglicherweise ab
```

Nachdem das Passwort eingegeben wurde (standardmäßig *vision*), besteht eine Verbindung zur Kamera. Es folgt eine kurze Erklärung der wichtigsten Befehle, um sich auf der Kamera zurecht zu finden.

“**cd**”
cd ohne zusätzliche Anweisung bringt einen zurück ins Home-Verzeichnis.
Hier: C:\Users\Nutzer

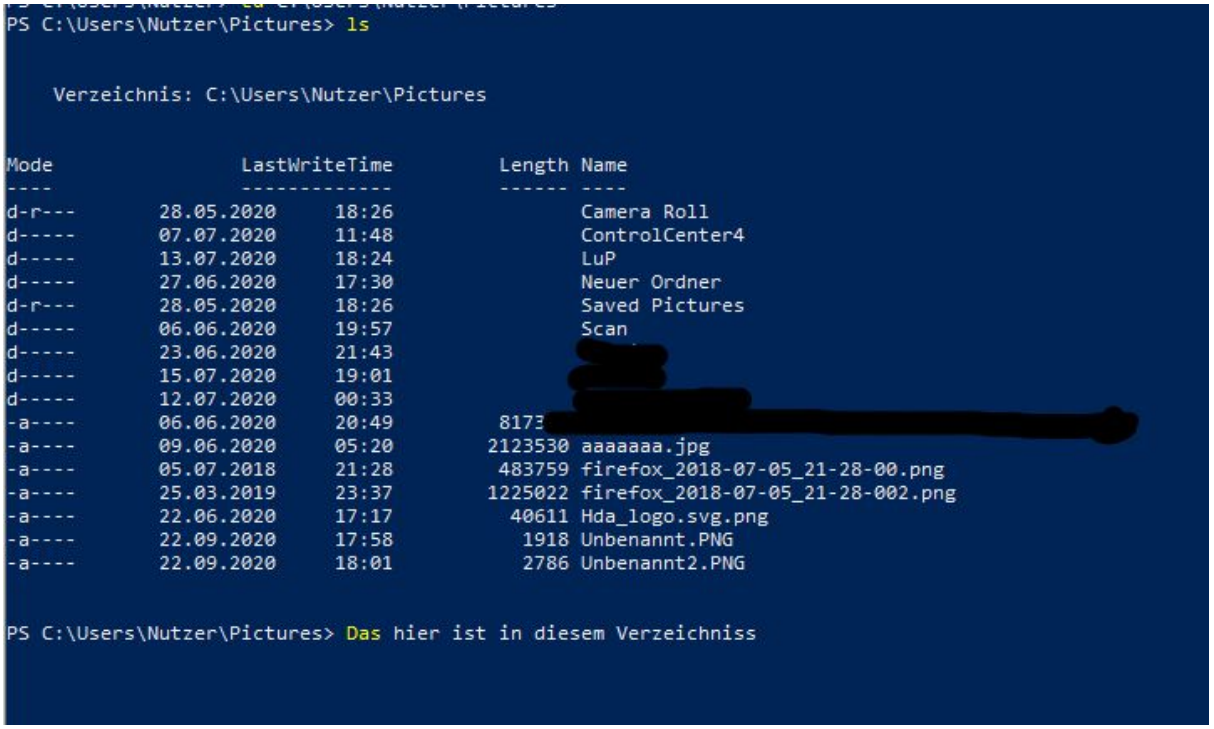
```
PS C:\Users\Nutzer> cd
PS C:\Users\Nutzer>  <--- Hier befinden wir uns
```

cd *Verzeichnis*
Wird hinter **cd** noch ein Verzeichnis angegeben, wechselt man in dieses Verzeichnis.
(mit “**cd ..**” wechselt man ins vorherige)

```
PS C:\Users\Nutzer> cd C:\Users\Nutzer\Pictures
PS C:\Users\Nutzer\Pictures>
>> Jetzt sind wir im Picture verzeichniss
```

ls

Mit dem Befehl “**ls**” lässt sich ausgeben, was sich im aktuellen Verzeichnis befindet.
Gibt man zusätzlich noch ein Verzeichnis ein, wird ausgegeben, was sich in dem besagten Verzeichnis befindet.



(Das sind nur Beispiele, auf der kamera heißen schon die Grund Verzeichnisse anders! Das Prinzip aber bleibt.)

Zum Testen der Kamera:
Im Verzeichnis **bin** befindet sich die ausführbare Datei **“LiveImageWeb”.** Diese wird mit folgendem ausgeführt:

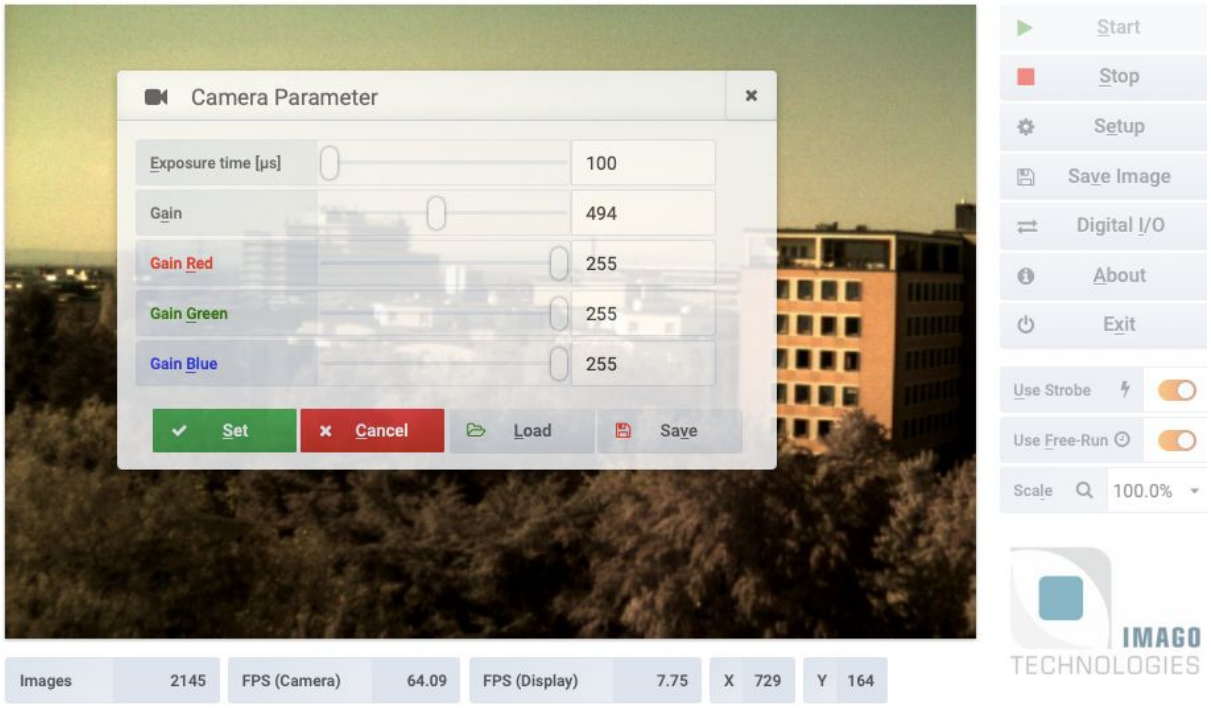
sudo ./LiveImageWeb

```
For more details, please visit https://support.apple.com/kb/HT208050.
(base) [redacted]$ ssh visionsensor@141.100.188.72
visionsensor@141.100.188.72's password:
Linux VSPV-DEB9_1_3a_AV1_0_1 4.9.59-rt23-visionsensor-pv-1.0.2.0 #14 Wed Aug 14
11:09:12 CEST 2019 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Aug  9 03:24:09 2020 from 141.100.182.75
visionsensor@VSPV-DEB9_1_3a_AV1_0_1:~$ cd bin
visionsensor@VSPV-DEB9_1_3a_AV1_0_1:~/bin$ sudo ./LiveImageWeb
[sudo] password for visionsensor:
Live Image (Web Interface)
TCP Port: 80
Authentication disabled
Press <enter> to stop the server
```

Nachdem die Datei **LiveImageWeb** ausgeführt wurde, kann in jedem beliebigen Browser die IP-Adresse der Kamera eingegeben werden. Wenn jetzt der Button START gedrückt wird, ist das Livebild der Kamera zu sehen. Unter Umständen muss jetzt im Bereich SETUP noch die Belichtungszeit angepasst werden (z.B. wenn das Bild weiß ist).



Falls nach einem Befehl die Meldung *unknown command* zurückgegeben wird, sollte es nochmal mit einem *sudo* vor dem Kommando getestet werden.

Unser Projekt ist auf der uns gestellten Kamera unter dem Pfad */home/visionsensor/projects/SmartCam3/* gespeichert. Der Code ist unter *SmartCam3/src/* zu finden. Die ausführbare Datei (**SmartCam3.out**) befindet sich unter *SmartCam3/bin/ARM/Debug.*

Visual Studio

Visual Studio kann als Entwicklungsumgebung verwendet werden. Vorteile sind eine nützliche Remote Kontrolle sowie ein Tutorial von IMAGO zur Verknüpfung.
Um einen selbstgeschriebenen Code auf der Kamera laufen lassen zu können, muss Visual Studio mit der Kamera verknüpft werden.

Verknüpfen von Visual Studio mit VisionSensorPV für Remote Kontrolle

In unserem Fall wurde Visual Studio (2019) benutzt um über Remote Kontrolle, auch von z.B. zuhause arbeiten zu können. Das bereits erwähnte Tutorial ist hierzu äußerst nützlich. Hierbei ist es zu Empfehlen das Projekt sofort auf ARM umzustellen, da es sonst zu Schwierigkeiten kommen könnte.

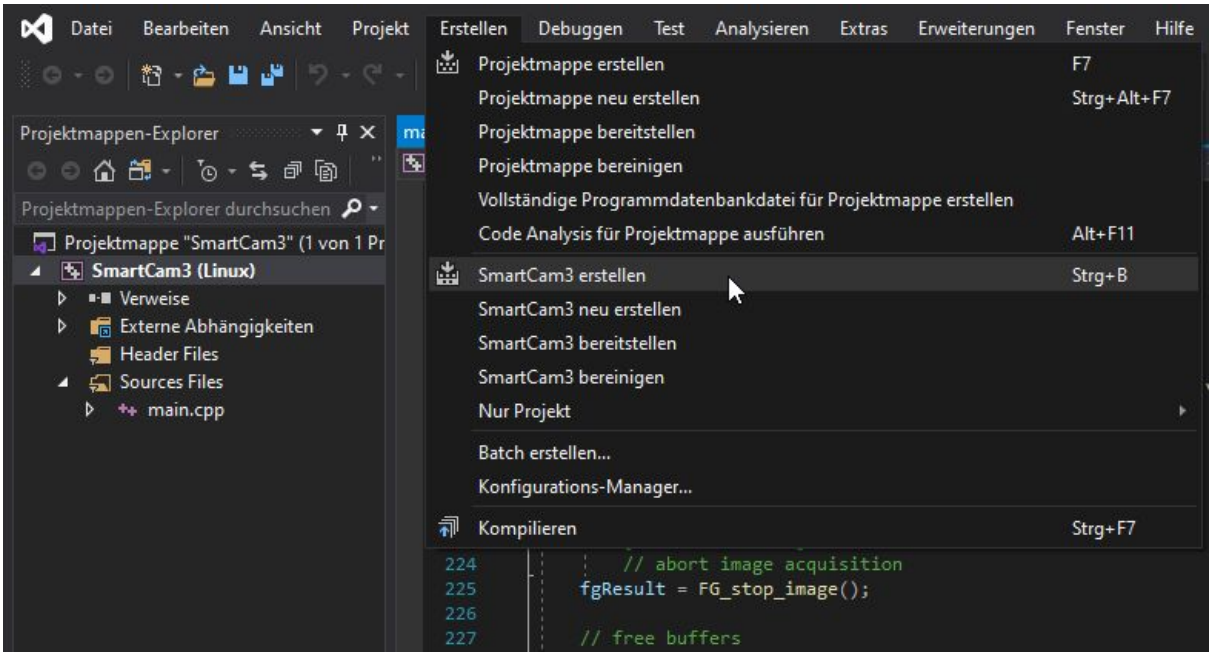
OpenCV (4.4.0) ist bereits auf der Kamera instaliert. Dennoch müssen die Include/Libary Paths zum eigenem VS Projekt hinzugefügt werden. C/C++ | Allgemein | Zusätzliche Includeverzeichnisse
/*usr/local/include/opencv4/* bzw. Linker | Input | Bibliotheksabhänigkeiten
opencv_core;opencv_imgcodecs;opencv_dnn;opencv_imgproc
Das Tutorial für die opencv Nutzung von Imago ist trotzdem zu empfehlen, da noch Verschiedenes mit Visual Studio verknüpft werden müssen.

Programm starten

Nun sollte sich ein eigenes VS Projekt mit eigenem Verzeichnis auf der Kamera befinden. Das bedeutet auch, dass das Projekt noch leer ist. Ein Beispielprojekt ist in dem Imago “example” Verzeichnis. Als Alternative kann auch der unten beschriebene Code verwendet werden. Dafür müssen sich allerdings alle erforderlichen Bibliotheken und Dateien auf der Kamera befinden.

Das Starten mit dem VS Debugger ist nicht unbedingt zu empfehlen, da der Code entweder manuell oder beim Testen von Code ggf. durch ein Error mittendrinnen gestoppt wird und die Zeile *fgResult = FG_uninstall_camera();* nicht ausgeführt wird. Dies führt zu einem SEGMENTATION FAULT und die Smart Cam muss mit *sudo reboot* neu gestartet werden.

Das kann man allerdings umgehen, indem man das Projekt auf die Kamera mit Erstellen | Projekt-Erstellen lädt und falls erfolgreich erstellt, von der Kamera startet.



cd “Verzeichnis in dem .out Datei ist”
sudo ./ProjektName.out
Das Programm kann mit strg+c gestoppt werden.

Neuronales Netzwerk vorbereiten:

Das neuronale Netzwerk zur Handgestenerkennung ist auf folgender Seite zu finden: https://github.com/filipefborba/HandRecognition. Die relevanten Teile dabei sind die Dateien “handrecognition_model.h5” sowie “project3.ipynb”. Die .h5 Datei ist das model. Darin ist das vortrainierte Netzwerk zu finden. In dem Jupyter Notebook “project3.ipynb” ist zu sehen, wie dieses Netzwerk trainiert und getestet wird. Daraus kann man sich die Teile raussuchen,

die man zum Verwenden des NNs benötigt. Zum Verständnis ist hier ein Code, der Bilder aus der Webcam ausliest und durch das NN auswertet:

```
import cv2
import matplotlib.pyplot as plt
from tensorflow import keras
import numpy as np

# import model
model = keras.models.load_model('handrecognition_model.h5')
# get camera frame
camera = cv2.VideoCapture(0)
_, frame = camera.read()
# convert frame to grayvalue image
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
camera.release()
# all the possible gestures
prediction_strings = ['Thumbs down', 'Palm (Horizontal)',
                      'L', 'Fist (Horizontal)', 'Fist (Vertical)',
                      'Thumbs up', 'Index', 'OK', 'Palm
(Vertical)',
                      'C']
# resize frame to size of the frames the model is trained with
frame = cv2.resize(frame, (320, 120))
# set threshold to avoid bright backround areas
frame[frame < np.max(frame)*0.6] = 0
# put the frame into an array (so you can have more frames the
same time)
frames = [frame]
# convert frame into an array
frames = np.array(frames, dtype="uint8")
# reshape the array into an more dimansional array
frames = frames.reshape(1, 120, 320, 1)
# put the frames into the model for predictions
prediction = model.predict(frames)
# print the prediction for our frame
print(prediction_strings[np.argmax(prediction)])
# print the propabilities for every possible gesture
for i, percentage in enumerate(prediction[0]):
   print('%s: %3.2f%%' % (prediction_strings[i], percentage*100))
# print the frame

plt.imshow(frames[0], 'gray')
```

Die Ausgabe könnte zum Beispiel so aussehen:

```
C
Thumbs down: 0.01%
Palm (Horizontal): 0.00%
L: 0.00%
Fist (Horizontal): 0.00%
Fist (Vertical): 0.00%
Thumbs up: 0.00%
Index: 0.00%
OK: 0.02%
Palm (Vertical): 0.00%
C: 99.95%
```

```
Out[48]: <matplotlib.image.AxesImage at 0x7f9cbfaa6080>
```

Um bessere Ergebnisse zu erzielen, ist es auch gut zu wissen wie die Gesten mit denen das NN trainiert wurde überhaupt aussehen. Die Datenbank für diese Bilder ist unter folgendem Link zu finden: https://www.kaggle.com/gti-upm/leapgestrecog.

Da es nicht möglich ist die Bibliothek “Tensorflow” auf der Kamera zu installieren (die Kamera hat nur ein 32bit System und die Bibliothek benötigt ein 64bit System), ist es notwendig das model in ein Dateiformat umzuformen, welches von OpenCV ausgelesen und verwendet werden kann. Das bedeutet wir konvertieren die .h5 Datei in eine .pb Datei. Allerdings benötigt OpenCV noch eine Konfigurationsdatei (.pbtxt) um mit dem Model etwas anfangen zu können. Diese beiden Dateien werden mit folgendem Code erstellt:

Zum Erstellen des Tensor Flow Models (.pb):

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.python.framework.convert_to_constants import
convert_variables_to_constants_v2
import numpy as np
print(tf.executing_eagerly())
#path of the directory where you want to save your model
frozen_out_path = ''
# name of the .pb file
frozen_graph_filename = "frozen_graph"
model = keras.models.load_model('handrecognition_model.h5')
# Convert Keras model to ConcreteFunction
```

```
full_model = tf.function(lambda x: model(x))
full_model = full_model.get_concrete_function(
   tf.TensorSpec(model.inputs[0].shape, model.inputs[0].dtype))
# Get frozen ConcreteFunction
frozen_func = convert_variables_to_constants_v2(full_model)
frozen_func.graph.as_graph_def()
layers = [op.name for op in frozen_func.graph.get_operations()]
print("-" * 60)
print("Frozen model layers: ")
for layer in layers:
   print(layer)
print("-" * 60)
print("Frozen model inputs: ")
print(frozen_func.inputs)
print("Frozen model outputs: ")
print(frozen_func.outputs)
# Save frozen graph to disk
tf.io.write_graph(graph_or_graph_def=frozen_func.graph,
                  logdir=frozen_out_path,
                  name=f"{frozen_graph_filename}.pb",
                  as_text=False)
# Save its text representation
tf.io.write_graph(graph_or_graph_def=frozen_func.graph,
                  logdir=frozen_out_path,
                  name=f"{frozen_graph_filename}.pbtxt",
                  as_text=True)
```

Zum Erstellen der Konfigurationsdatei (.pbtxt):

```
# Read the graph.
with tf.io.gfile.GFile('frozen_graph.pb', 'rb') as f:
   graph_def = tf.compat.v1.GraphDef()
   graph_def.ParseFromString(f.read())

# Remove Const nodes.
for i in reversed(range(len(graph_def.node))):
   if graph_def.node[i].op == 'Const':
       del graph_def.node[i]
   for attr in ['T', 'data_format', 'Tshape', 'N', 'Tidx', 'Tdim',
                'use_cudnn_on_gpu', 'Index', 'Tperm',
'is_training',
                'Tpaddings']:
       if attr in graph_def.node[i].attr:
           del graph_def.node[i].attr[attr]

# Save as text.
tf.compat.v1.train.write_graph(graph_def, "", "model.pbtxt",
as_text=True)
```

Code der SmartCam3.cpp

Um diesen Code ausführen zu können sind die Bibliotheken “OpenCV”, “iostream” sowie “iomanip” notwendig. Das neuronale Netz sollte sich in demselben Ordner des Codes befinden.
Im Folgenden wird der Code zur Handgestenerkennung abschnittsweise erklärt.

In dem ersten Teil werden alle nötigen Bibliotheken eingebunden. Außerdem wird hier eine Funktion definiert, welche die Bildaufnahme-Schleife beendet, wenn das Programm ein “Interrupt Signal” (z.B durch STRG - C) erhält.

```
#include <stdio.h>
#include <signal.h>

#include "FG_CameraInterface.h"

#include <opencv4/opencv2/core.hpp>
#include <opencv4/opencv2/imgcodecs.hpp>
#include <opencv4/opencv2/imgproc.hpp>
#include <opencv2/dnn.hpp>
#include <iostream>
#include <iomanip>
#include "unistd.h"


#define NUM_BUFFERS 4   // number of image buffers used for the
acquisition queue


int isRunning = 1;

// Signal handler for stopping the program
static void sigint_handler(int signo)
{
   if (signo == SIGINT)
       isRunning = 0;
}
```

Ab diesem Teil beginnt die main-Funktion. Hier wird zu Beginn das Neuronale Netz in Form einer Tensorflow Datei mit ihrer Konfigurationsdatei eingelesen. Außerdem wird ein Fehler ausgegeben, falls das Neuronale Netz leer sein sollte. Darauf folgen einige Definitionen von Parametern, welche zur Bildaufnahme dienen. Nun wird die Kamera initialisiert und es werden weitere Parameter definiert, welche zur Bildaufnahme dienen.

```
int main()
```

```
{

   auto hand_gesture_net =
cv::dnn::readNetFromTensorflow("hand_recognition_model.pb",
"hand_recognition_model.pbtxt");
   int model_empty = hand_gesture_net.empty();
   printf("Read Model... Model empty: %d\n", model_empty);
   FG_IMAGE imageList[NUM_BUFFERS];
   UINT32 fgResult;
   char ErrText[512];
   UINT32 x1, y1, x2, y2;
   UINT32 width, height;
   enum eFG_PIXEL_TYPE pixel_type;
   UINT32 shuttertime = 25000;



   // register signal handler for CTRL-C:
   signal(SIGINT, sigint_handler);


   // initialize the camera using automatic type detection
   fgResult =
FG_install_camera(FG_CAMERA_TYPE_X_X_IMAGO_Vxx_AUTO);
   if (fgResult != FG_ERROR_CODE_NoError)
   {
       FG_get_last_error(ErrText, sizeof(ErrText));
       printf("FG_install_camera() failed! \n - ErrorCode: %d \n -
ErrorText '%s'", fgResult, ErrText);
       return -1;
   }

fgResult = FG_set_trigger_mode(FG_TRIGGER_MODE_FREERUN);

// get default AOI and pixel type:
fgResult = FG_get_aoi(&x1, &y1, &x2, &y2, &pixel_type);
// set new pixel type:
fgResult = FG_set_aoi(x1, y1, x2, y2, FG_PIXEL_TYPE_Y_8);
//! [set parameters]


width = x2 - x1 + 1;
height = y2 - y1 + 1;
printf("Sensor size: %u x %u\n", width, height);
//! [buffer allocation]
   // allocate image buffers and them to the queue
for (UINT32 i = 0; i < NUM_BUFFERS; i++)
{
   fgResult = FG_alloc_image(&imageList[i]);
   fgResult = FG_append_image(&imageList[i]);
```

```
}

printf("Starte Programm. Zum Beenden \"Strg+C\".\n");
printf("Stelle die Belichtungszeit ein...\n");
```

Nun startet die tatsächliche Bildaufnahme. Diese while - Schleife läuft solange durch, bis es durch die erste Funktion des Codes unterbrochen wird. Ab der vierten Zeile wird die Belichtungszeit an die Umgebung angepasst. In den Kommentaren des Codes ist das genaue Vorgehen hiervon näher erläutert.

```
while (isRunning)
{
   FG_IMAGE currentImage;
   UINT32 stepcountermin = 0; // Zähler wie oft die
Belichtungszeit hintereinander erhöht wurde
   UINT32 stepcountermax = 0;  // Zähler wie oft die
Belichtungszeit hintereinander verringert wurde
   UINT32 shuttertime_temp = shuttertime - 100; // Prüfvariable ob
Belichtungszeit sich im letzten Durchgang geändert hat

// [Belichtungszeit loop startet]
while (shuttertime_temp != shuttertime && isRunning) //wenn die
Belichtungszeit im letzten durchlauf nicht angepasst wurde wird
die Schleife abgebrochen.
{
   shuttertime_temp = shuttertime;
   fgResult = FG_set_shutter_time(shuttertime); // Belichtungszeit
wird übergeben
   fgResult = FG_get_image(&currentImage, 10000); // Bild wird
aufgenommen mit 1000 mikrosekunden timeout
   cv::Mat image = cv::Mat(height, width, CV_8UC1,
currentImage.pixel_ptr, cv::Mat::AUTO_STEP); // aufgenommenes Bild
wird in eine  cv-Mat übertragen.

   UINT32 countermax = 0; // Anzahl Pixel mit Grauwert >= 175
   UINT32 countermin = 0; // Anzahl Pixel mit Grauwert <= 50

   // [Pixel werden einzeln überprüft und eingeordnet]
   for (UINT32 y = 0; y < height; y++)
   {
       for (UINT32 x = 0; x < width; x++)
       {
           if (image.ptr(y)[x] >= 175)
               countermax += 1;
           if (image.ptr(y)[x] <= 50)
               countermin += 1;
       }
   }
```

```
   // ![Pixel werden einzeln überprüft und eingeordnet]

   FG_append_image(&currentImage); // Buffer wird wieder
freigegeben
   if (countermax >= 0.001 * height * width && countermin >= 0.001
* width * height) //Überprüft ob genug Pixel innerhalb der Grenzen
liegen, um die aktuelle Belichtungszeit zu verwenden
   {
       printf("Belichtungszeit final eingestellt auf %u
Mikrosekunden..\n\n", shuttertime);
   }
   else if (countermax >= 0.001 * height * width) // passt die
Belichtungszeit nach unten an, wenn zu viele Pixel zu hell sind.
(vergrößert Schritte je öfter es hintereinander verringert)
   {
       if (stepcountermax >= 6 && shuttertime >= 10000)
       {
           shuttertime -= 10000;
           stepcountermax += 1;
       }
       else if (stepcountermax >= 3 && shuttertime > 1000)
       {
           shuttertime -= 1000;
           stepcountermax += 1;
       }
       else if (stepcountermax < 3 || shuttertime <= 1000)
       {
           shuttertime -= 100;
           stepcountermax += 1;
           stepcountermin = 0;
       }
   }
   else if (countermin >= 0.001 * width * height) // passt die
Belichtungszeit nach oben an, wenn zu viele Pixel zu dunkel sind.
(vergrößert Schritte je öfter es hintereinander verringert)
   {
       if (stepcountermin >= 6)
       {
           shuttertime += 10000;
           stepcountermin += 1;
       }
       else if (stepcountermin >= 3)
       {
           shuttertime += 1000;
           stepcountermin += 1;
       }
       else if (stepcountermax < 3)
       {
           shuttertime += 100;
           stepcountermin += 1;
```

```
            stepcountermax = 0;
        }
    }
}
```

In diesem Abschnitt kommt das neuronale Netzwerk zum Einsatz. Nachdem die Belichtungszeit angepasst wird, wird das aktuelle Bild abgegriffen und in die Variable *image* geschrieben, welche vom Typ cv::Mat ist. Das Bild wird mithilfe eines OpenCV Rechtecks auf die richtige Größe für das neuronale Netz zurecht geschnitten. Da so viel des Bildes wie möglich genutzt werden sollte, haben wir die doppelte Größe der für das neuronale Netz verwendet. Deshalb muss es mit der Funktion *resize* skaliert werden. In den letzten drei Zeilen dieses Abschnitts wird das Bild in das neuronale Netz gesteckt und das Ergebnis des Netzes wird zur weiteren Auswertung in einen double vector mit dem Namen *output* kopiert.

```
if (fgResult == FG_ERROR_CODE_NoError)
{
   fgResult = FG_set_shutter_time(shuttertime); // microseconds
   fgResult = FG_get_image(&currentImage, 1000);
   cv::Mat image = cv::Mat(height, width, CV_8UC1,
currentImage.pixel_ptr, cv::Mat::AUTO_STEP);


   cv::Rect image_crop(48, 119, 640, 240);
   cv::Mat cropped_image = image(image_crop);
   printf("Bild aufgenommen.\n\n");
   cv::Mat detection_image;
   cv::resize(cropped_image, detection_image, { 320, 120 });
   auto input = cv::dnn::blobFromImage(detection_image);
   hand_gesture_net.setInput(input);
   auto output = hand_gesture_net.forward();
```

Das Output des neuronalen Netzes sind die verschiedenen Wahrscheinlichkeiten für jede der zehn Handgesten. Diese werden in diesem Abschnitt verwendet und so formatiert, dass man auf den ersten Blick auf das Terminal sieht, wie gut das Netz abgeschnitten hat.

```
  std::vector<double> outputs;
  output.copyTo(outputs);
       int count = 0;
       std::vector<std::string> gestures = { "Thumbs Down", "Palm
Horizontal", "L", "Fist Horizontal", "Fist Vertical", "Thumbs Up",
"Index", "Ok", "Palm Vertical", "C" };
       double  maxlik = 0;
       int idx = 0;
       for (int i = 0; i < outputs.size() - 1; i++){
           if (outputs[i] >= maxlik)
           {
               maxlik = outputs[i];
```

```
                idx = i;
            }
        }
        std::cout << std::string(50, '_') << "\n\n" << std::endl;

        for (auto percentage : outputs) {
            std::cout << gestures[count] << std::string((16 -
gestures[count].length()), ' ') << ":" << std::setw(25) <<
percentage * 100 << " %\n";
            count++;
        }

        std::cout << std::string(50, '_') << "\n\n" << std::endl;

        std::cout << "Wahrscheinlichste Geste: \"" << gestures[idx]
<< "\" mit " << maxlik * 100 << " % Sicherheit\n\n" << std::endl;
        // add the buffer again to the free queue again
        FG_append_image(&currentImage);
    }
    else if (fgResult == FG_ERROR_CODE_BrokenImage)
    {
        // the image is valid, but the contents are broken or
incomplete, add the buffer again or release it
        printf("FG_get_image() returned a broken image\n");
        FG_append_image(&currentImage);
    }
    else if (fgResult == FG_ERROR_CODE_GrabTimeOut)
    {
        // timout occured, the image is invalid
        printf("FG_get_image() timeout occured\n");
    }
    else
    {
        // other error occured, the image is invalid
        FG_get_last_error(ErrText, sizeof(ErrText));
        printf("FG_get_image() failed! \n - ErrorCode: %d \n -
ErrorText '%s'", fgResult, ErrText);
        break;
    }
    std::cout << "Naechste Bildaufnahme in 7 Sekunden...\n" +
std::string(50, '*') << "\n"<< std::endl;
    sleep(7);
}
```

Dieser letzte Abschnitt des Codes ist zum Abbruch der Bildaufnahme notwendig. Er lässt den Speicher wieder frei und schließt die Kamera.

```
// abort image acquisition
    fgResult = FG_stop_image();
```

```
    // free buffers
    for (UINT32 i = 0; i < NUM_BUFFERS; i++)
    {
        fgResult = FG_free_image(&imageList[i]);
    }

    // Close the camera
    fgResult = FG_uninstall_camera();

    return 0;
}
```

Fazit

Das Ziel war es, dass die Kamera Handgesten ohne Unterstützung eines Computers erkennen kann. Ein weiteres Ziel welches gesetzt wurde, war die vom neuronalen Netz erkannte Handgeste über einen Arduino auf einem LCD Display darzustellen.
Das erste Ziel wurde größtenteils erreicht. Das neuronale Netz läuft auf der Kamera auch ohne externe Unterstützung, allerdings ist die Ausgabe dessen noch in der Konsole.
Für das zweite Ziel müsste zunächst die Hürde auf der Software - Seite genommen werden. Es war zum Ende hin nicht möglich ein Arduino Projekt über Visual Studio auf der Kamera zum Laufen zu bringen. Vermutlich wird das von der Remote Kontrolle nicht unterstützt. Auf der Hardware - Seite ist alles soweit fertig, sodass wir nur noch über die Bibliothek von Imago die Digital Output Pins hätten ansprechen müssen.