Make:

KI für den ESP32: Neuronale Netze zur Bilddigitalisierung nutzen

25.04.2022 10:00 Uhr Josef Müller

Mit TensorFlow Lite und C++ lässt sich KI einsteigertauglich auf Embedded-Systemen implementieren. Wir zeigen, wie man das mit der ESP32CAM macht.

In diesem Artikel zeigen wir, wie man TensorFlow Lite auf einer ESP32-CAM zur Ziffernerkennung betreibt. Die Kernidee, nämlich die Verwendung von neuronalen Netzen zur Bilddigitalisierung, steckt in nur zwei zentralen Bibliotheken und relativ wenigen Zeilen selbst geschriebenem Code. Davon ausgehend lassen sich leicht eigene Projekte mit eingebetteter KI ableiten.

Bevor es jedoch an die Programmierung (oder an die Erklärung unseres Codes) geht, muss zunächst die Hardware und die Programmierumgebung vorbereitet werden.

ESP32-PROJEKTE

- Anleitung: Zugangskontrolle mit RFID selbst einrichten [1]
- Lebend-Mausefalle mit ESP32-CAM basteln [2]
- Neuronale Netze zur Bilderkennung mit Python trainieren [3]
- KI für den ESP32: Neuronales Netz trainieren für Strom-/Wasser-/Gaszähler-Daten [4]
- KI für den ESP32: Neuronale Netze zur Bilddigitalisierung nutzen [5]
- Make-Projekt: ESP32-Orgel mit Piano-Tastatur bauen [6]
- Make-Projekt: SMD-Löten mit dem Pizza-Ofen [7]
- Maker-Workshop: Der Weg zur Platine [8]
- LoRa: Netzunabhängige und stromsparende Türüberwachung bauen [9]
- Anleitung zum Strom sparen bei ESP-Mikrocontrollern [10]
- Smarthome-Firmware für ESP8266/32-Module sichern und flashen [11]
- ESP32 für Profis: Hardware ausreizen mit ESP-IDF [12]
- Visual Studio Code: Installation und Konfiguration für ESP IDF [13]

Typischerweise wird der ESP32 in DIY-Projekten in einem integrierten Modul eingesetzt, auf dem extern notwendige Beschaltungen wie WLAN-Antenne oder Spannungswandler bereits vorhanden sind.

KURZINFO

- Visual Studio Code installieren, konfigurieren und bedienen
- Neuronale Netze in TensorFlow Lite und C++ implementieren
- ESP32CAM flashen und neuronales Netz testen

Checkliste

Zeitaufwand: 1 Stunde

Kosten: 10 Euro

Material

- **ESP32CAM** (AI Thinker)
- **USB2Serial-Wandler** plus Jumper-Kabel
- Alternativ: **Development Board mit CH340**

Die ESP32CAM hat noch folgende Komponenten integriert:

1. SD-Kartenleser
2. PSRAM (8 MByte, davon 4 MByte effektiv nutzbar)

3. 4 MByte Flash
4. OV2640 Kameraschnittstelle und -modul
5. LED-Flashlight

Insbesondere die Nutzung von SD-Karten und der erweiterte Speicher sind für neuronale Netze sehr von Vorteil. Grundsätzlich lassen sich die neuronalen Netze auch vollständig in der Firmware abbilden. Wenn man jedoch die Größe von neuronalen Netzen mit mehr als 100.000 Knoten anschaut, wird schnell klar, dass die 512 KByte SRAM bald an ihre Grenzen kommen. Hier hilft das PSRAM weiter, da es auf einfache Weise den Arbeitsspeicher um 4 MByte erweitert. Hier sei schonmal vermerkt, dass die Unterstützung von PSRAM in der Konfiguration des Compilers aktiviert werden muss.

Die SD-Karte dient vor allem als Speichermedium für Beispielbilder und auch zum dynamischen Laden von neuronalen Netzen während des Programmablaufs. Beides wird im Quellcode gezeigt.

Auf das eigentliche Kameramodul, wie man damit zur Laufzeit Bilder aufnimmt und Ziffern erkennt, gehen wir in diesem Artikel ganz am Ende ein. Die Erklärungen zum Steuern der Kamera und dem Aufnehmen und Normalisieren eines Bildes unter C++ würde den Rahmen dieses Artikels sprengen.

Das ESP32CAM-Modul inklusive der Kamera bekommt man schon für ca. 10 EUR bei den einschlägigen Versendern. Bei günstigen Angeboten findet man in Foren immer wieder mal den Hinweis, dass doch nur 2 MByte PSRAM verbaut sind oder die Kamera Schwierigkeiten macht. Der PSRAM wird daher auch zu Beginn des Programms abgeprüft und gegebenenfalls eine Warnung ausgegeben.

[14]

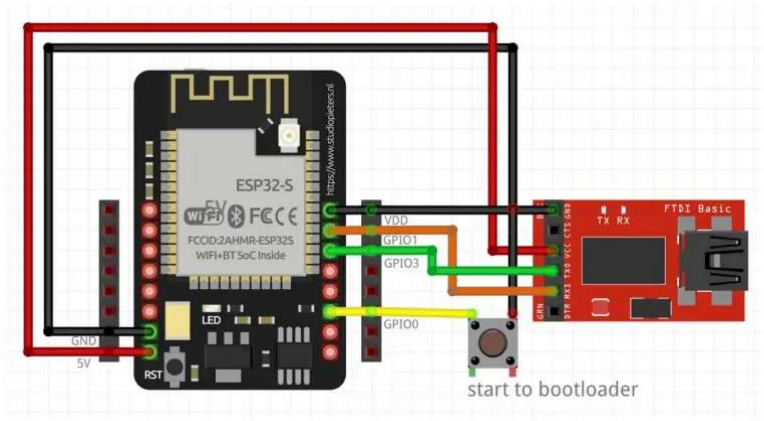
Vorbereitung

Programmier-Interface

Um die Firmware auf den ESP32 zu flashen, ist eine Programmierschnittstelle notwendig. Diese verbindet die UART-Schnittstelle des ESP32 mit der USB-Schnittstelle des Rechners. Dabei gibt es im Wesentlichen zwei Möglichkeiten: Zum ersten gibt es häufig das ESP32-CAM Modul bereits in Kombination mit einem Adapter, der eine Mini-USB-Schnittstelle und darüber auch die Stromversorgung enthält.

Man findet es häufig unter der Bezeichnung *ESP32-CAM-MB*. Zum zweiten kann man auch einen FTDI-Adapter mit den entsprechenden GPIOs verbinden. Dann muss man natürlich auch die Spannungsversorgung sicherstellen. Die Kommunikation ist an GPIO1 und GPIO3 angeschlossen.

Um den ESP32 im Downloadmodus zu starten, muss der GPIO0 während des Starts auf Masse-Potential (GND) gelegt werden – im Bild angedeutet durch einen Taster.



Die ESP32CAM lässt sich mit einem USB-zu-seriell-Konverter programmieren. Einfacher macht es das Development Board für die ESP32CAM, das eine Spannungsversorgung und einen Button für Pin IO0 mitbringt.

Beim ESP32-CAM-MB ist dazu ein extra Schalter vorhanden. Wenn der ESP32 autark läuft oder per WLAN/Bluetooth-Schnittstelle angesteuert wird, ist nur noch eine Stromversorgung notwendig. Es wird eine 5-V-Versorgung empfohlen, da 3,3 V teilweise nicht stabil laufen. Damit ist die Hardware für die Programmierung vorbereitet.

SD-Karte

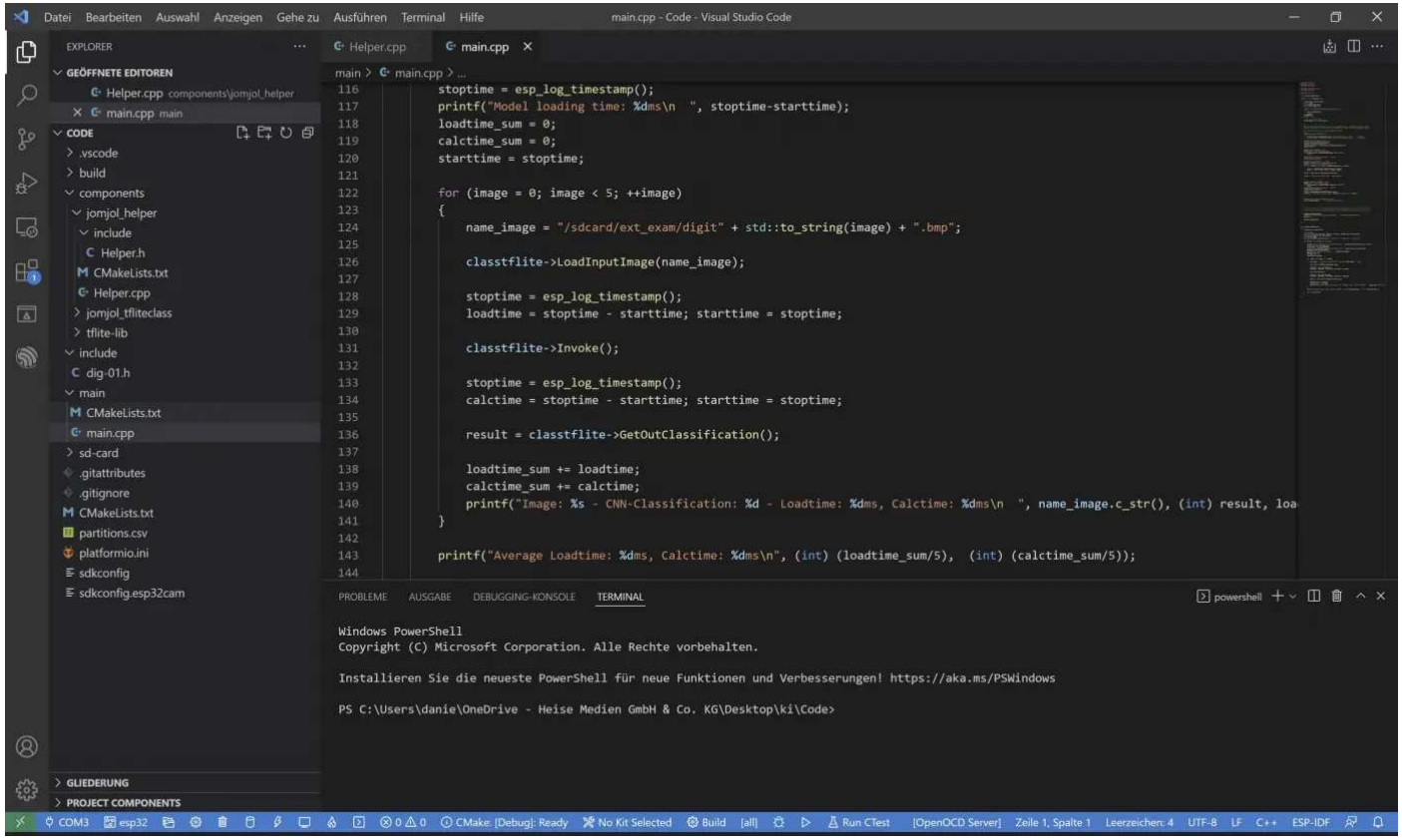
Der SD-Kartenleser nimmt normale Micro-SD-Karten auf. Eigentlich keine große Sache. Leider hat sich jedoch im *AI-on-the-Edge*-Projekt gezeigt, dass nicht alle SD-Karten kompatibel sind. Insbesondere große SD-Karten mit 32/64 GByte in Kombination mit der internen 1-Kanal-Datenverbindung zwischen ESP32 und Kartenleser sind problematisch. Daher wird hier standardmäßig die 4-Kanal-Verbindung verwendet, die dann aber auch den GPIO12 und 13 belegt. Auch der GPIO4 wird verwendet, was eine Doppelbelegung mit dem LED-Flashlight bedeutet. Daher leuchtet im Beispielprogramm das Flashlight bei Lesezugriffen auch auf.

Die SD-Karte muss FAT32 formatiert sein und darf nur eine einzige Partition enthalten. Wenn die Karte sich nicht initialisieren lässt, dann gibt es eine entsprechende Fehlermeldung auf der Konsole. Leider gibt es, wenn auch selten, das Problem, dass die Karte sich initialisieren lässt, dann aber die Dateien nicht lesbar sind. In diesem Fall hilft nur eine andere SD-Karte, und zwar möglichst klein – 4 GB-Karten machen typischerweise keine Probleme.

Programmierumgebung

Die Programmierung des ESP32 erfolgt in C/C++, da darin auch die TensorFlow-Lite-Bibliotheken für Mikrocontroller programmiert sind. Als Programmierumgebung insbesondere auch für einfache bis mittel komplexe Projekte wird häufig die Arduino-IDE verwendet. In diesem Projekt wird jedoch eine deutlich leistungsfähigere und umfangreichere Umgebung benötigt.

Als Editor kommt der kostenlose **Visual Studio Code (VSCode) (Download) [15]** von Microsoft zum Einsatz, der sich mit zahlreichen Plugins, Compilern und Interpretern erweitern lässt. Mit dem ebenfalls kostenlosen ESP-IDF-Plug-in lassen sich alle Produkte des Herstellers Espressif in VSCode programmieren und flashen. Allerdings stehen hier die einsteigerfreundlichen Bibliotheken und Makros wie in der Arduino-Welt nicht mehr zu Verfügung, die kompliziertere Code-Teile weitestgehend abstrahieren.



MIt Visual Studio Code behält man in komplexeren Projekten eine bessere Übersicht als mit der Arduino IDE.

Die Anleitung zur Installation, Konfiguration und Bedienung von Visual Studio Code haben wir in einem **Artikel "Visual Studio Code: Installation und Konfiguration für ESP IDF" ausgelagert [16]**, zu dem es zusätzlich ein Video-Tutorial gibt.

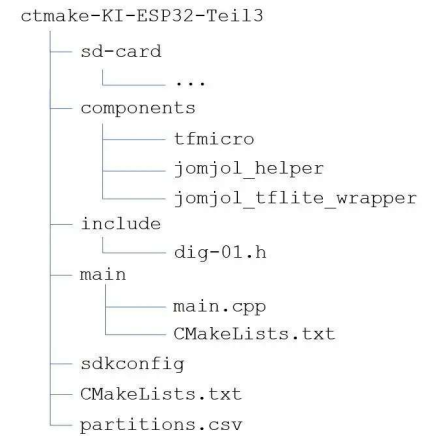
Das ESP-IDF-Plugin installiert und konfiguriert die Toolchain für das Übersetzen des C-Codes, das Linken und das Flashen.

Eine detaillierte Installationsanleitung und ein Video mit einem ersten einfachen Demobeispielprojekt findet sich **in unserem begleitenden Artikel [17]**. Die hier verwendete Toolchain von Espressif wird in der Version 4.3 verwendet.

Programmstruktur

Wenn nach der Installation das Beispielprogramm läuft, steht dem eigentlichen Projekt nichts mehr im Weg. Zunächst benötigen wir Code **aus dem GitHub-Repository [18]**. Die wesentliche Programmstruktur ist in der Grafik gezeigt. Neben dem Hauptverzeichnis gibt es vier Unterverzeichnisse:

- *sd-card*
- *components*
- *include*
- *main*



Überblick über die Programmstruktur des Projektes

Das Verzeichnis *sd-card* wird nicht zum Kompilieren benötigt, sondern enthält die Dateien, die während der Laufzeit auf der SD-Karte benötigt werden. Der Inhalt des Verzeichnisses muss auf die vorbereitete SD-Karte

kopieren werden. Im Verzeichnis *main* findet sich das eigentliche Hauptprogramm. Dazu später mehr, ebenso zum Verzeichnis *include*, welches eine direkt im Code einbindbare Definition des neuronalen Netzes enthält.

Wichtig ist das Verzeichnis *component*. Dort wird der Code für alle verwendeten Komponenten abgelegt. Konkret sind das in diesem Projekt drei Bibliotheken:

jomjol_helper: Diese dient im Wesentlichen dazu, das Hauptprogramm so übersichtlich wie möglich zu halten und typische Standardaufgaben in Funktionen auszulagern, welche dann im Hauptprogramm einfach aufgerufen werden können. Darin finden sich zum Beispiel Hilfsprozeduren zum Initialisieren der SD-Karte oder zum Prüfen des PSRAMs.

jomjol_tfliteclass: Hierin ist eine eigene Klasse in C++ implementiert, die die Ansteuerung der TensorFlow-Lite-Komponenten kapselt und alle sich wiederholenden Abläufe zum Laden, Initialisieren und Verwenden von neuronalen Netzen enthält. In dieser Klasse sind auch einige Hilfsfunktionen implementiert, um die Bilder zu laden oder Informationen über die Ein- und Ausgangslayer des Netzes zu bekommen. Mit dieser Art Wrapper-Klasse lässt sich die komplexe dritte Bibliothek, die eigentliche Bibliothek für die neuronalen Netze, in einer sehr einfachen und schlanken Art und Weise nutzen. Mehr dazu gleich.

tflite-lib: Dies ist der eigentliche Code von TensorFlow. Dieser Code kann aus dem offiziellen TensorFlow-Lite-Code abgeleitet und für die ESP-IDF-Umgebung angepasst werden. Glücklicherweise hat Espressif in seinen eigenen Beispielprogrammen für den ESP32 das bereits für uns erledigt. Dort findet man die Bibliothek im Verzeichnis *component*. Dieses Verzeichnis wird von Espressif regelmäßig aktualisiert. In unserem Projekt-Verzeichnis findet sich die Kopie des **ESP-Github-Repos [19]** vom Stand Anfang 2022.

Neben diesen Verzeichnissen sind noch drei Dateien von wesentlicher Bedeutung: *CMakeLists.txt*, *sdkconfig* und *partitions.csv.*

Wichtige Dateien

CMakeLists.txt

Erstere findet sich in jedem Hauptverzeichnis des Quellcodes und steuert die Details für den Compiler für die jeweilige Komponente. Das sollte erst mal so übernommen werden. Wer schon mal unter Linux C/C++-Code übersetzt hat, ist womöglich mit dem Tool `make` in Berührung gekommen. Es steuert anhand der zum jeweiligen Projekt gehörenden Datei *Makefile* alle Befehle, Parameter und Schritte beim Übersetzen und Linken. Auch die Arduino-IDE verwendet im Hintergrund das make-System, um aus Sketchen Binärdateien zu bauen.

`CMake` geht einen Schritt weiter und macht Projekte unabhängig vom Betriebssystem und der Entwicklungsumgebung. Es erstellt aus den Schritten in der Datei *CMakeLists.txt* die für das verwendete Framework passenden Anleitungen, beispielsweise Makefiles. Hier zeigen wir als Beispiel die Datei *CMakeLists.txt* im Verzeichnis *main*, die dem Compiler notwendige Schritte vorgibt.

```
idf_component_register(SRCS \$
         {CMAKE_SOURCE_DIR}/main/main.cpp`

                   `INCLUDE_DIRS \$
         {CMAKE_SOURCE_DIR}/include`

                   `REQUIRES jomjol_tfliteclass jomjol_helper)
```

Dem Compiler müssen verschiedene Dinge vorgegeben werden. Dies geschieht durch den Aufruf von `idf_component_register` mit folgenden Informationen respektive Schlüsselworten: der Pfad zum Quelltext des Hauptprogramms *main.cpp* – definiert über `SRCS`, das Verzeichnis der Header-Dateien (`INCLUDE_DIRS`) und weitere wichtige Bibliotheken (`REQUIRES`). Letzteres verweist auf weitere benötigte Komponenten in dem Verzeichnis *component*.

Hier sind es zum einen die Hilfskomponenten `jomjol_helper` und `jomjol_tfliteclass`, die die Ansteuerung des neuronalen Netzes übernehmen. Die Bibliothek `tflite-lib` muss hier nicht angegeben

werden, da sie nicht direkt aufgerufen wird, sondern indirekt über die `jomjol_tfliteclass` eingebunden wird, die über ein eigenes CMakeFile übersetzt wird (siehe *CMakeList.txt* im jeweiligen Unterverzeichnis). Kompilierinformationen können auch deutlich komplexer ausfallen – etwa das Makefile für die Komponente *tflite-lib* im entsprechenden Unterverzeichnis.

sdkconfig

Wichtig ist auch die Datei *sdkconfig* im Hauptverzeichnis. Dort werden alle Compiler-spezifischen Einstellungen für den ESP32 gespeichert. Zur Konfiguration klickt man in VSCode das Rädchen im linken Bereich des unteren Menübands. Es öffnet sich ein neuer Tab (was beim ersten Mal einige Zeit dauern kann) mit dem Namen *SDK Configuration Editor*. Hier sind zwei Einstellungen wichtig: Aktivierung von PSRAM und die Verwendung einer eigenen Partitionstabelle. Das ist in unserem Projekt bereits korrekt vorkonfiguriert. Wenn man jedoch die ESP-IDF-Version ändert, aktualisiert oder das Projekt auf eine andere Entwicklungsumgebung wie *PlatformIO* umzieht, müssen diese Einstellungen überprüft und gegebenenfalls wieder angepasst werden.

Partitions.csv

In der Datei *partitions.csv* wird eine benutzerdefinierte Partitionierung des Flashspeichers definiert, um die 4 MByte vollständig auszunutzen. Dies wird im Abschnitt `Serial flasher config` im Editor der *sdkconfig* angegeben. Üblicherweise haben ESP32-Module mit 4 MB Speicher eine Aufteilung in fünf Partitionen, wozu auch Bereiche für das Update der Firmware per WLAN (OTA) vorgesehen sind. Das verbraucht viel Speicher, den wir in unserem Projekt benötigen, da wir das neuronale Netz direkt in den Programmcode implementieren. Die richtige Partitionierung ist in der Datei schon voreingestellt.

ESP32-Programmierung

Nun sind alle Vorbereitungen getroffen und es geht an die eigentliche Programmierung. Im Prinzip kann man mit der Hilfsbibliothek *jomjol_tflitewrapper* in fünf Schritten einen Programmteil zur Nutzung von neuronalen Netzen erstellen:

1. Erzeugen eines Objekts der Wrapper-Klasse *CTfLiteClass*
2. Laden der Models

3. Laden eines Bildes
4. Berechnen des neuronalen Netzes
5. Auslesen der Ergebnisse

Diese Schritte sind in der Klasse als eigene Methoden implementiert; ihre Namen sind im Prinzip selbsterklärend, siehe folgende Tabelle. Eine detaillierte Erklärung finden Sie in *CTfLiteClass.h*.

Methoden der Wrapper-Klasse

Name	Funktion
LoadModelFromFile	Netz von SD-Karte laden
LoadModelFromCharArray	Netz aus Header laden
LoadInputImage	Bild von SD-Karte laden
Invoke	Berechnen der Ausgabe (Inferencing)
GetOutClassification	Ausgabe-Neuronen ausgeben
GetOutputValue	Neuron mit dem höchsten Wert ausgeben
GetInputDimension	Größe des Input-Layer ausgeben
GetOutputDimension	Größe der Ausgabeschicht ausgeben

Ihre Arbeitsweise wollen wir in den nächsten Abschnitten der Reihe nach durchgehen und um zusätzliche Informationen ergänzen. Diese Schritte sind im Hauptprogramm in der `main`-Funktion abgebildet.

Bibliotheken

Damit alle Funktionen zur Verfügung stehen, müssen zunächst zwei Bibliotheken geladen werden:

```
#include
    "CTfLiteClass.h"

#include
    "Helper.h"
```

Als erster Schritt werden mit Hilfe von Funktionen aus der Helper-Bibliothek die SD-Karte eingebunden und die Verfügbarkeit des PSRAMs geprüft. Dahinter findet sich eine rudimentäre Fehlerbehandlung. Details dazu finden sich direkt in den Funktionen im Code (*main.cpp*) beziehungsweise in den Ausgaben auf der Konsole.

Im weiteren müssen wir eine Instanz der Klasse `CTfLiteClass` erzeugen, um ein neuronales Netz aufzubauen:

```
neuralnetwork = new CTfLiteClass;
```

Dabei werden im Hintergrund die Variablen initialisiert und ein Speicherbereich reserviert, in dem die *TfLite*-Bibliothek anschließend ihren Daten und Strukturen speichert. Dessen Größe hängen von der Größe des neuronalen Netzes ab. Hier sind zunächst 600 KByte voreingestellt. Das ist eher großzügig und ausreichend, auch für größere Netzwerke.

Neuronales Netz

Laden des Modells

Als Nächstes wird die *tflite*-Beschreibung des neuronalen Netzwerkes geladen. Dort kommt jetzt das Trainingsergebnis des zweiten Teils zum Einsatz. Man hat zwei Möglichkeiten, die Modellbeschreibung des

neuronalen Netzes einzubinden. Entweder lädt man die *tflite*-Datei dynamisch während des Programmablaufs von der SD-Karte:

```
if
(
!neuralnetwork->
LoadModelFromFile
(
"/sdcard/dig-01.tfl"
)
;
// dynamisch
return
;
```

oder man bindet es direkt als `char`-Array aus der automatisch erzeugten Header-Datei statisch ein:

```
if
(
!neuralnetwork->
LoadModelCharArray
(tflite_model)
)
;
// statisch
```

```
        return
        ;
```

Beim statischen Einbinden muss man die in unserem **vorangegangenen Artikel "KI für den ESP32: Neuronales Netz trainieren für Strom-/Wasser-/Gaszähler-Daten" [20]** erzeugte Headerdatei während bzw. vor dem Übersetzungsvorgang einbinden. Dies geschieht hier durch die Zeile:

```
#include"dig-01.h"
```

Darin ist das Modell in einem `char`-Array namens `tflite_model` definiert, dass dann direkt eingebunden wird. Das Laden direkt aus einem Array hat den Vorteil, dass man kein Dateisystem benötigt und damit entsprechend einfachere Hardware einsetzen kann.

Der Vorteil des dynamischen Ladens von der SD-Karte liegt zum einen in der Speicherverwaltung, denn er wird erst dynamisch während des Programmablaufes im PSRAM allokiert und kann für andere Aufgaben wieder freigegeben werden. Zum Zweiten kann man verschiedene neuronale Netze zur Laufzeit nach Bedarf laden. Man benötigt dafür aber ein Speichermedium, wie hier die SD-Karte. Es lassen sich Daten aber auch über das Netzwerk oder von anderen Speichermedien nachladen.

Sind die *tflite*-Daten geladen, müssen die Strukturen und Speicheranforderungen im vorher reservierten Speicherbereich angelegt und initialisiert werden. Dies übernimmt die interne Funktion `MakeAllocate()` innerhalb der Ladefunktion. Im Code findet man noch zwei weitere Befehle (`GetInputDimension()` und `GetOutputDimension()`), die die Struktur des geladenen neuronalen Netzes auslesen und wiedergeben: am Eingang sind es hier drei (20 × 32 ×3) Dimensionen und am Ausgang nur eine Dimension (mit 11 Neuronen).

Laden des Bildes

Jetzt ist das neuronale Netz bereit, um Bilder zu bewerten. Dazu wird zunächst das Bild geladen – hier von der SD-Karte. In realen Anwendungen könnte hier auch ein Ausschnitt des Kamerabildes des integrierten OV2640-Kameramoduls geladen werden.

```
image_file =
          "/sdcard/digit3a.jpg"
          ;
          if
          (
          !neuralnetwork->
          LoadInputImage
          (image_file.
          c_str
          (
          )
          )
          );
   return;
```

Hier können sowohl jpg- wie auch bmp-Bilder verwendet werden. Das Bild wird über eine Hilfsbibliothek geladen, die die Bildpunkte Pixel für Pixel an die Eingangsneuronen des ersten Layers anlegt. Dies erfordert etwas C-Zeigerarithmetik und ist innerhalb der *jomjol_tfliteclass*-Bibliothek implementiert.

Im nächsten Schritt wird das neuronale Netz berechnet. Das übernimmt eine einzige Funktion:

```
neuralnetwork-> Invoke (  ) ;
```

Dies ist der rechenintensivste Prozess und kann einen Moment dauern.

Auslesen der Ergebnisse

Als letzten Schritt gibt das Programm die Ergebnisse über die serielle Schnittstelle aus. Um sich diese anzeigen zu lassen, muss man in der unteren Leiste den Punkt *ESP IDF Monitor Device* anklicken. Die Ausgabe des Ergebnisses erfolgt in `main()` auf zwei Arten:

```
        for
        (
        int _output =
        0
        ; _output < AnzahlOutputNeurons; _output++
        )
        {
    result = neuralnetwork->
        GetOutputValue
        (_output)
        ;
        printf
        (
        "   Neuron #%d: %.1f\n"
        , _output, result)
        ;
        }
```

In der Schleife wird über die Funktion `GetOutputValue(_output)` der Wert jedes einzelnen Neurons ausgelesen und ausgegeben. Die gesuchte Klassifizierung ergibt sich aus dem Neuron mit dem höchsten Aktivitätswert. Will man nur das Neuron mit dem höchsten Wert, genügt die Funktion `GetOutClassification()`, die wir für ein weiteres Bild nutzen:

```
result = neuralnetwork-> GetOutClassification ( );
```

Damit ist die Basis für die Anwendung zur Bildverarbeitung gelegt. Um diese wenigen Zeilen Code herum kann man jetzt die weiteren Strukturen zur Bereitstellung der Bilder und dem Darstellen der Ergebnisse entwickeln. Zum Beispiel könnte man ein Bild direkt mit der vorhandenen Kamera aufnehmen und Abschnitte daraus an das neuronale Netz übergeben; über Standardbibliotheken auf die Eingangsgröße des neuronalen Netzes skalieren und dann klassifizieren lassen. Das Ergebnis könnte man in einer Webapplikation darstellen oder auch auf die SD-Karte speichern.

Wie in unserem Tutorial zur Installation von VSCode und der ESP-IDF müssen Sie den gesamten Code durch Anklicken von *ESP IDF Build Project* übersetzen. Das kann einige Zeit in Anspruch nehmen. Anschließend lädt man das Binary auf die Kamera, steckt die SD-Karte mit den *TFlite*-Modellen und den Bilddateien und resettet. Mit einem Klick auf *ESP IDF Monitor Device* öffnet sich ein serieller Monitor, mit dem sich die Ausgabe des ESP verfolgen lässt.

```
SD-Card using 4-line connection mode. GPIO12/13 is used by SD-Card.
I (1432) gpio: GPIO[13]| InputEn: 0| OutputEn: 1| OpenDrain: 0| Pullup: 0| Pulldown: 0| Intr:0
Name: SU08G
Type: SDHC/SDXC
Speed: 20 MHz
Size: 7580MB
PSRAM initialisiert - freier verfügbarer Speicher: 4192151

Lade tflite-Model

Größe Eingangs-Layer:
Anzahl Eingangsdimensionen: 3
```

```
  Größe Dimension 1: 32
  Größe Dimension 2: 20
  Größe Dimension 3: 3


Größe Ausgangs-Layer:
Anzahl Ausgangsdimensionen: 1
  Größe Dimension 1: 11

Lade Bilddaten...
Berechne neuronales Netz ...

Frage Ergebnis ab ...
Filename: /sdcard/digit3a.jpg
 Einzelne Output-Neuronen:
   Neuron #0: 0.0
   Neuron #1: 0.0
   Neuron #2: 0.0
   Neuron #3: 1.0
   Neuron #4: 0.0
   Neuron #5: 0.0
   Neuron #6: 0.0
   Neuron #7: 0.0
   Neuron #8: 0.0
   Neuron #9: 0.0
   Neuron #10: 0.0
  CNN-Klassifizierung: 3


Lade Bilddaten...
Berechne neuronales Netz ...

Frage Ergebnis ab ...
/sdcard/digit1.jpg: CNN-Klassifizierung: 1

Herzlichen Glückwunsch!
```

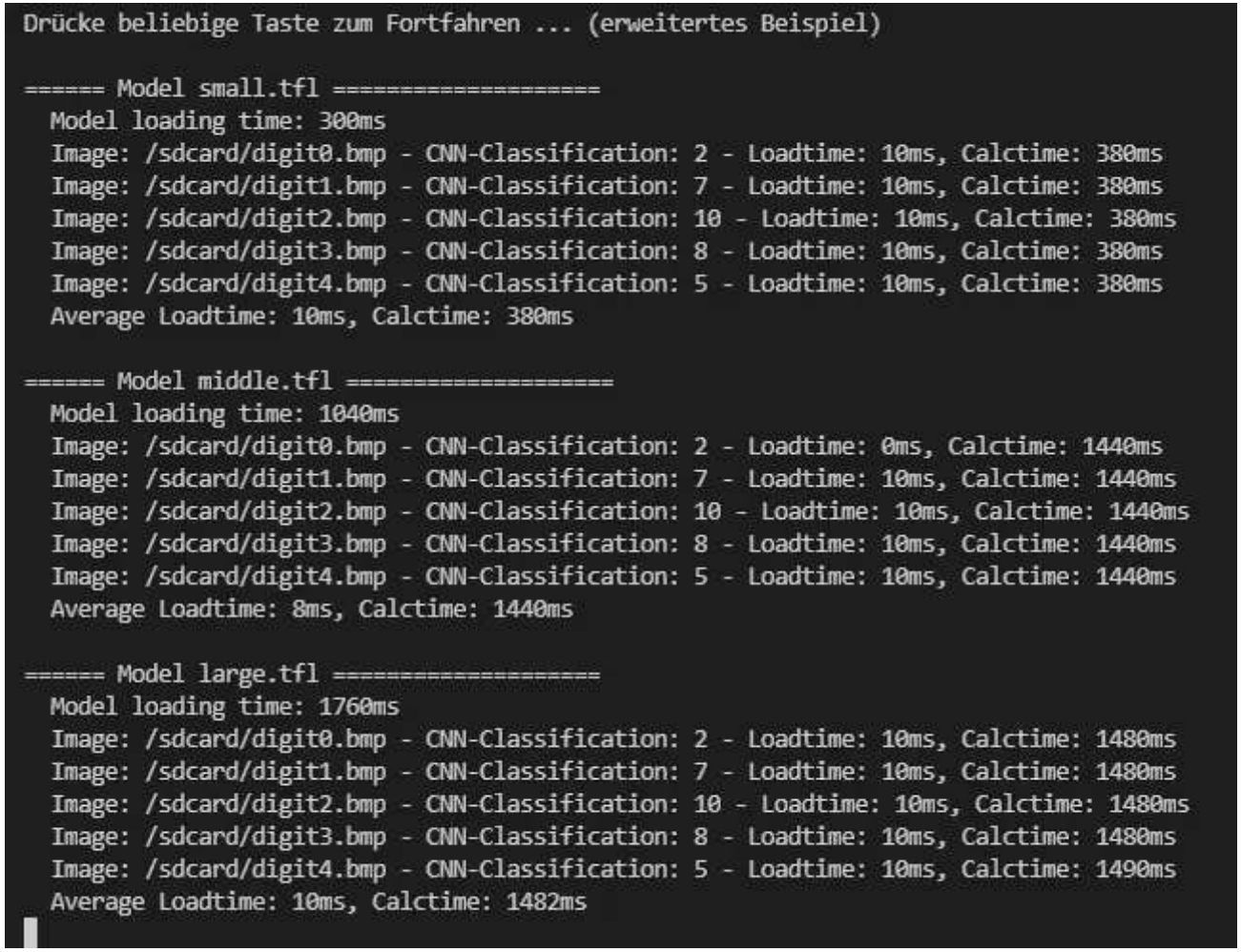
Über die serielle Schnittstelle gibt die ESP32CAM ihren Status und ihre Ergebnisse aus.

Dieses Beispiel zeigt, wie man neuronale Netze, die man in einer beliebigen Umgebung trainiert hat, sehr einfach auf dem ESP32 anwenden kann. Das Laden der neuronalen Netze direkt aus dem *tflite*-Format ermöglicht auch

eine flexible Anpassung und Tests verschiedener Netzkonfigurationen. So könnte man in einer zweiten Schleife auch unterschiedlich Netzgrößen und Strukturen verwenden und die Ergebnisse oder Performance auf dem ESP32 vergleichen.

Aufgrund des erweiterten Speichers ist es möglich, *tflite*-Files, die größer als 1 MB sind und nahezu eine Million Parameter enthalten, zu laden und laufen zu lassen. Mit diesen wenigen Zeilen Code hat man schon die Berechnung eines selbst trainierten neuronalen Netzes sowie die Anwendung auf beliebige Bilder ermöglicht.

Im Hauptprogramm wird nach dem ersten Teil durch einen Tastendruck ein zweites, etwas umfangreicheres Beispiel gestartet. In diesem werden der Reihe nach drei neuronale Netze von der SD-Karte geladen, welche eine unterschiedliche Größe haben. In einer Schleife werden dann fünf Bitmaps berechnet respektive erkannt.

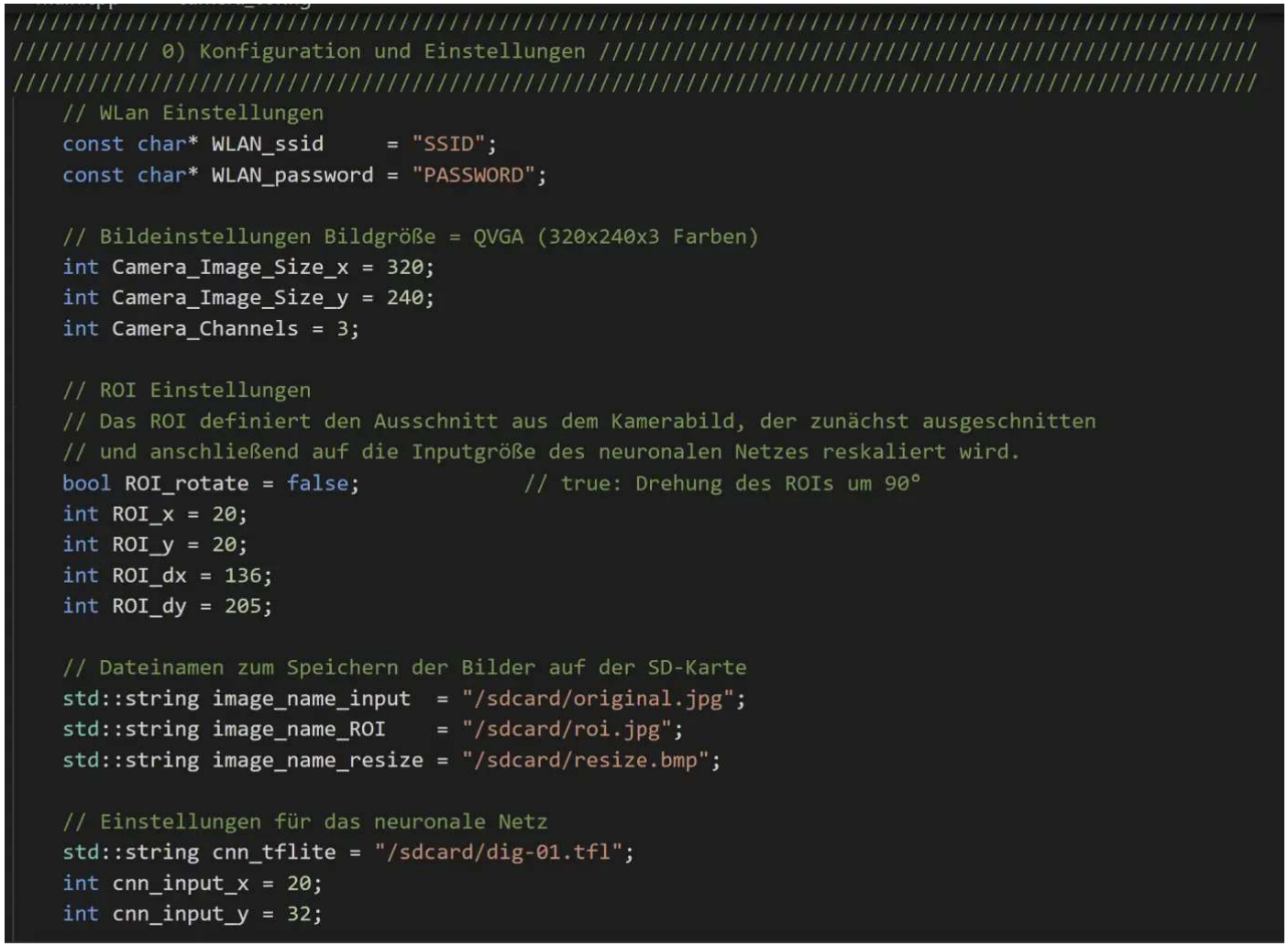


Verschiedene neuronale Netze bei ihrer Arbeit

Sowohl die Zeit zum Laden des neuronalen Netzes wie auch die Berechnungszeit der einzelnen Bilder wird angezeigt. In diesem Beispiel werden zum einen die Vorteile des dynamischen Ladens von neuronalen Netzen gezeigt, wie auch durch die unterschiedlich großen Netze ein Gefühl für die Berechnungsdauer von unterschiedlichen Netzgrößen entwickelt.

Live-Erkennung

Ganz bewusst haben wir aus didaktischen Gründen im ersten Programm auf die Verwendung der Kamera verzichtet, da neben der einfachen Bildaufnahme noch wichtige Zusatzschritte notwendig sind, damit ein reproduzierbares und zufriedenstellendes Ergebnis herauskommt. Dazu gehören unter anderem: Ausrichten des Kamerabildes, Identifikation des Bildausschnitts (in der Regel nicht das gesamte Bild) und Skalierung des Bildes.



```
////////////////////////////////////////////////////////////////////////////////////////////////////
/////////// 0) Konfiguration und Einstellungen /////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////
    // WLan Einstellungen
    const char* WLAN_ssid     = "SSID";
    const char* WLAN_password = "PASSWORD";

    // Bildeinstellungen Bildgröße = QVGA (320x240x3 Farben)
    int Camera_Image_Size_x = 320;
    int Camera_Image_Size_y = 240;
    int Camera_Channels = 3;

    // ROI Einstellungen
    // Das ROI definiert den Ausschnitt aus dem Kamerabild, der zunächst ausgeschnitten
    // und anschließend auf die Inputgröße des neuronalen Netzes reskaliert wird.
    bool ROI_rotate = false;             // true: Drehung des ROIs um 90°
    int ROI_x = 20;
    int ROI_y = 20;
    int ROI_dx = 136;
    int ROI_dy = 205;

    // Dateinamen zum Speichern der Bilder auf der SD-Karte
    std::string image_name_input  = "/sdcard/original.jpg";
    std::string image_name_ROI    = "/sdcard/roi.jpg";
    std::string image_name_resize = "/sdcard/resize.bmp";

    // Einstellungen für das neuronale Netz
    std::string cnn_tflite = "/sdcard/dig-01.tfl";
    int cnn_input_x = 20;
    int cnn_input_y = 32;
```

Für das Beispiel der Live-Erkennung gibt es einen Konfigurationsbereich in main.cpp.

Insbesondere das Ausrichten und Identifizieren der sogenannten *Region of Interest* (ROI), welches den Bildausschnitt für das neuronale Netz ausmacht, ist ein ganz zentraler und teilweise sehr aufwendiger Algorithmus. Das wird einem klar, wenn man berücksichtigt, dass die neuronalen Netze typischerweise mit einer Verschiebung von bis zu 20 % trainiert wurde, aber bei einer Größe der ROIs von typischerweise 30 - 100 Pixel dies auch nur 6 bis 20 Pixel Toleranz ergibt.

Beim ESP32 kommt noch hinzu, dass er per se erst mal keine grafische Benutzeroberfläche hat, auf der man die Ausrichtung der Bilder kontrollieren und justieren kann. Um dennoch die Kombination der OV2640 Kamera des ESP32CAM-Moduls und der neuronalen Erkennung in einem Gesamtsystem zu demonstrieren, stellen wir ein Programm-Beispiel bereit, das alle Elemente dazu enthält und auch für das Alignment eine rudimentäre Lösung bietet.

Dazu sind allerdings deutlich mehr und umfangreichere Bibliotheken notwendig als im ersten Beispiel. Eine detaillierte Erklärung im Detail finden Sie im Programmcode. **Der Quellcode findet sich ebenfalls auf Github [21]** und kann in derselben Umgebung wie auch das erste Beispiel kompiliert werden.

Ausgangspunkt des zweiten Programms ist ein sehr rudimentäres Beispiel aus dem ESP32CAM-Repository von Espressif. Darin wird die Kamera initialisiert und immer wieder ein Bild aufgenommen, jedoch ohne Speicherung. Wir ergänzen dieses Beispiel durch einen WLAN-Client, einen einfachen http-Server, um die aufgenommenen Bilder anzuzeigen sowie Funktionen zur Aufbereitung der aufgenommenen Bilder, damit sie vom neuronalen Netz verarbeitet werden können.

Zunächst werden im ersten Teil des Programms (*main.cpp*) die Variablen initialisiert und damit der Ablauf gesteuert (*0 Konfiguration und Einstellungen*). Hier ist insbesondere die Einstellungen für das WLAN wichtig, damit der ESP32 sich verbinden kann und man im Browser über den http-Server die Kamerabilder kontrollieren kann.

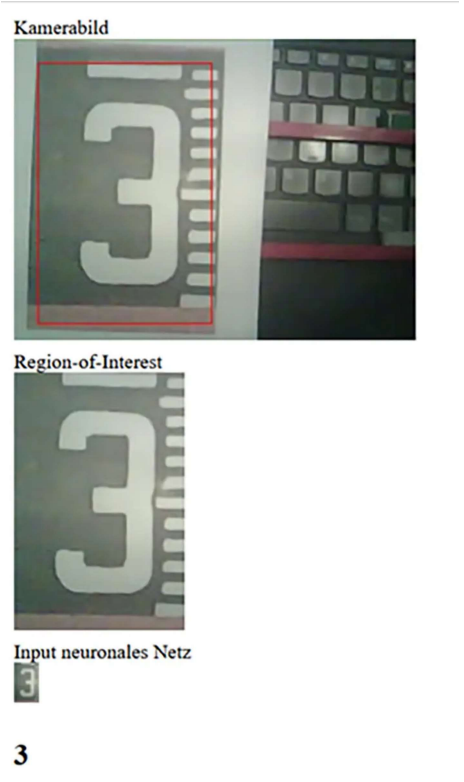
Letzte Schritte
Bildverarbeitung

Der Prozess besteht aus drei Schritten. Die Bildaufnahme: Hier wird das Bild von der Kamera geladen und über eine zusätzliche Bildverarbeitungsbibliothek auf der SD-Karte gespeichert. Die Bildgröße ist auf QVGA festgelegt. In das Bild wird vor dem Speichern noch das ROI gezeichnet, welches im nächsten Schritt weiterverwendet wird. Die Größe und Position des ROIs wird im Konfigurationsteil festgelegt.

Im zweiten Schritt wird mit Hilfe von zusätzlichen Bildverarbeitungsbibliotheken (*CImage*, *stb_image*) aus dem Originalbild zunächst das ROI ausgeschnitten, ggf. um 90 Grad gedreht (konfigurierbar) und auf die Zielgröße für das neuronale Netz (20 x 32 Pixel) skaliert. Beide Bilder (Original ROI und reskaliertes ROI) werden auf der SD-Karte abgespeichert. Im dritten Schritt berechnet das neuronale Netz das Ergebnis und zeigt es auf dem Webserver an.

Das Alignment, also die richtige Orientierung des Bildes, wird nicht durch das Programm gelöst, sondern unter Zuhilfenahme des Anwenders und des http-Servers. Der Server beziehungsweise die ausgelieferte Webseite ist so eingestellt, dass sie sich alle 2 Sekunden selbst aktualisiert und das Bild sowie das Ergebnis darstellt.

Anhand des Bildes und mit Hilfe des eingezeichneten ROIs kann der Nutzer die Kamera ausrichten und verschiedene Bilder testen. Beispielbilder sind in einer Word-Datei im Projektordner zum Ausdruck mitgeliefert. Der Webserver wird über seine IP-Adresse im LAN aufgerufen; diese wird über die Monitoringschnittstelle in VSCode zu Beginn jedes neuen Bildverarbeitungszyklusses angezeigt.

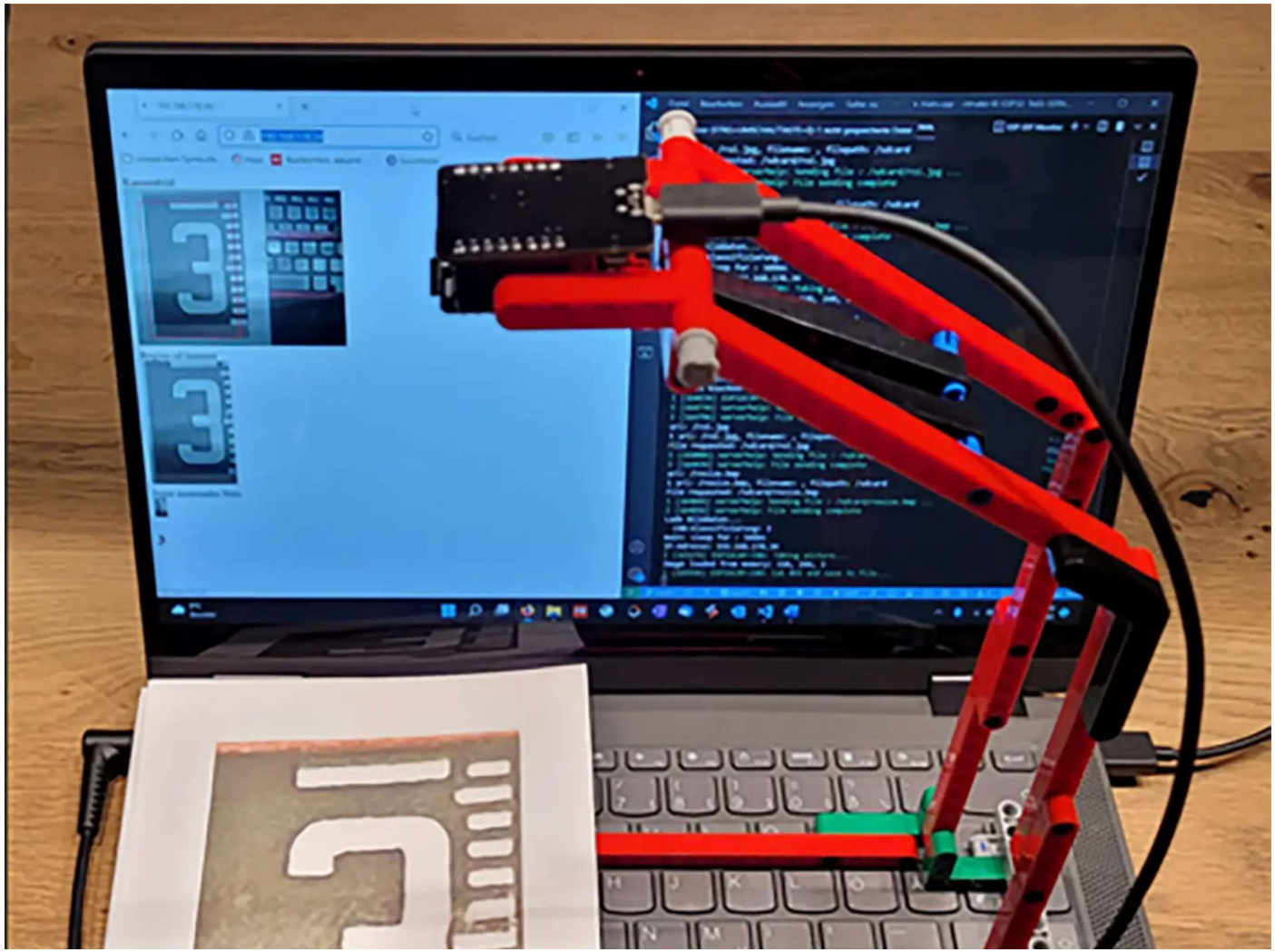


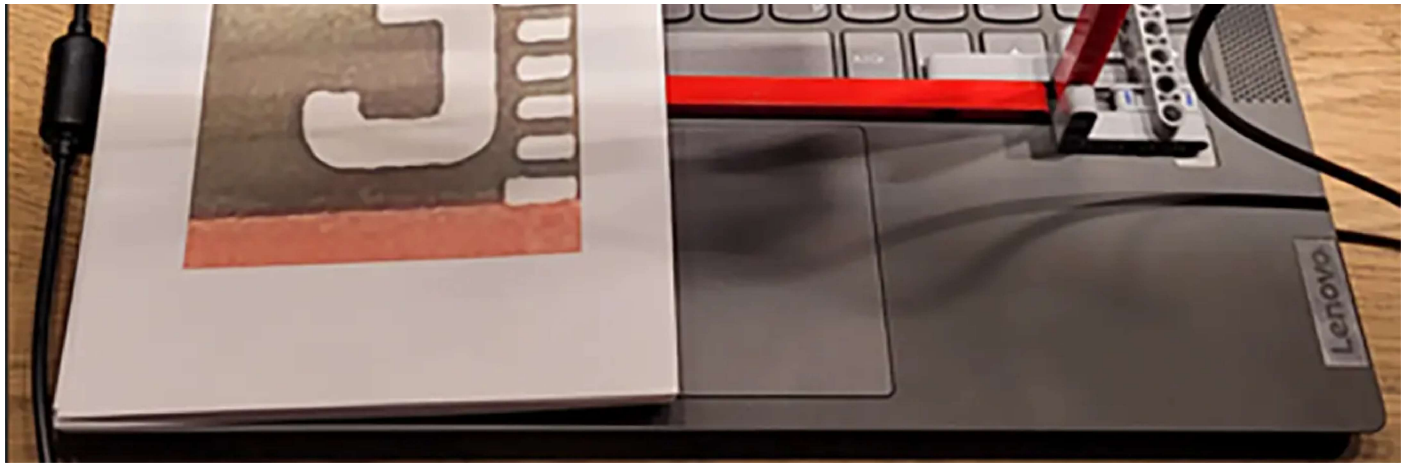
Die vom Webserver ausgelieferte Seite zeigt das Bild, die Region of Interest und die erkannte Ziffer an.

Der Server ist über die ESP-IDF eigenen Bibliotheken realisiert und in eine separate Komponente ausgelagert. Er enthält nur einen einzigen Handler, der die Anfragen je nach URL abarbeitet.

Inbetriebnahme

Um das ganze in Betrieb zu nehmen, muss zunächst die Konfiguration angepasst werden. Zunächst reicht es, die SSID und das Passwort des WLANs einzustellen, damit man den Server erreicht. Jetzt kann die Firmware kompiliert und auf das ESP32CAM-Modul geflasht werden. Als nächstes muss noch der Inhalt des Ordners sdcard auf die SD-Karte kopiert werden. Hier enthält er nur ein einziges File nämlich das tflite-File für das neuronale Netz. Jetzt kann der ESP32 in Betrieb genommen werden. Am besten startet man parallel den seriellen Monitor, um zu sehen, ob der ESP ohne Probleme startet. Dort ist auch die IP-Adresse abzulesen.





Der stabile Probeaufbau macht die Ausrichtung robuster und die Erkennung zuverlässiger.

Damit hoffen wir, dass wir Ihnen einen ersten Einblick in neuronale Netze und deren Verwendung auf einem günstigen Mikroprozessor gegeben haben. Viel Spaß bei eigenen Projekten und schreiben Sie uns! (**dab [22]**)

URL dieses Artikels:

https://www.heise.de/-7062278

Links in diesem Artikel:

[1] https://www.heise.de/ratgeber/ESP32-Projekt-Zugangskontrolle-mit-RFID-selbst-einrichten-7224274.html

[2] https://www.heise.de/ratgeber/Lebend-Mausefalle-mit-ESP32-CAM-basteln-7159932.html

[3] https://www.heise.de/hintergrund/Neuronale-Netze-zur-Bilderkennung-mit-Python-trainieren-6325534.html

[4] https://www.heise.de/hintergrund/Workshop-KI-fuer-den-ESP32-6345203.html

[5] https://www.heise.de/ratgeber/KI-fuer-den-ESP32-Neuronale-Netze-zur-Bilddigitalisierung-nutzen-7062278.html

[6] https://www.heise.de/ratgeber/Make-Projekt-ESP32-Orgel-mit-Piano-Tastatur-bauen-6323297.html

[7] https://www.heise.de/ratgeber/Make-Projekt-Loeten-im-Pizza-Ofen-6347219.html

[8] https://www.heise.de/ratgeber/Der-Weg-zur-Platine-Teil-1-6329981.html

[9] https://www.heise.de/ratgeber/LoRa-Netzunabhaengige-und-stromsparende-Tuerueberwachung-bauen-6163216.html

[10] https://www.heise.de/hintergrund/Anleitung-zum-Strom-sparen-bei-ESP-Mikrocontrollern-6131024.html

[11] https://www.heise.de/tests/Smarthome-Projekt-Vier-Loesungen-zum-Upgraden-von-Firmware-fuer-ESP-Module-4863989.html

[12] https://www.heise.de/ratgeber/ESP32-fuer-Profis-Hardware-ausreizen-mit-ESP-IDF-4488009.html

[13] https://www.heise.de/ratgeber/Visual-Studio-Code-Installation-und-Konfiguration-fuer-ESP-IDF-6656091.html

[14] https://www.heise.de/make/

[15] https://www.heise.de/download/product/visual-studio-code-96653/download

[16] https://www.heise.de/ratgeber/Visual-Studio-Code-Installation-und-Konfiguration-fuer-ESP-IDF-6656091.html

[17] https://www.heise.de/ratgeber/Visual-Studio-Code-Installation-und-Konfiguration-fuer-ESP-IDF-6656091.html

[18] https://github.com/makemagazinde

[19] https://github.com/MakeMagazinDE/ESP32-Special

[20] https://www.heise.de/hintergrund/Workshop-KI-fuer-den-ESP32-6345203.html

[21] https://github.com/MakeMagazinDE/ESP32-Special

[22] mailto:dab@ct.de