**Interactive Learning Tool for Children with Dyslexia using Arduino Mega and TFT Touch Screen**

**Table of Contents**

1. Abstract
2. Table of Contents
3. Table of Figures
4. Tables
5. Introduction
6. Project Equipment and Components
   1. Arduino Overview
   2. TFT Screen
7. Theory of Operation
8. Operation
9. The Code
   1. Python Description
   2. C++ Description

**Table of Figures**

* Figure 1: Arduino Mega Board
* Figure 2: TFT Touch Screen Display
* Figure 3: System Schematic Diagram
* Figure 4: Neural Network Architecture
* Figure 5: Sample Grid on TFT Screen
* Figure 6: ANN Prediction Display

**Tables**

* Table 1: Arduino Mega Specifications
* Table 2: TFT Screen Specifications
* Table 3: Neural Network Parameters
* Table 4: Example Input Data
* Table 5: Weights and Biases for ANN Layers

1. Abstract

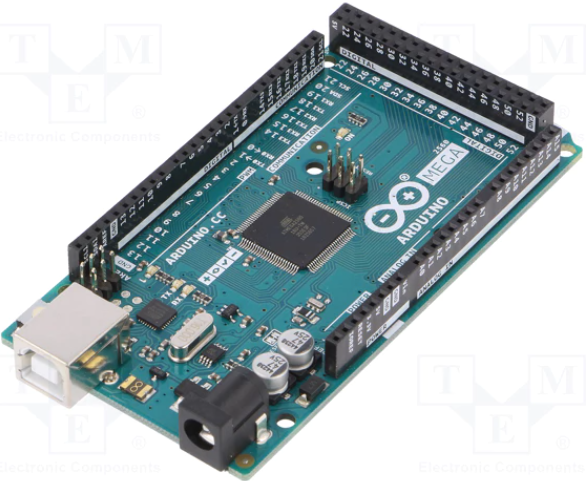
This project presents an interactive learning tool designed to assist children with dyslexia in improving their reading and writing skills. Utilizing an Arduino Mega and a TFT touch screen, the system incorporates artificial neural network (ANN) technology for digit recognition, providing an engaging and effective learning experience.

5. Introduction

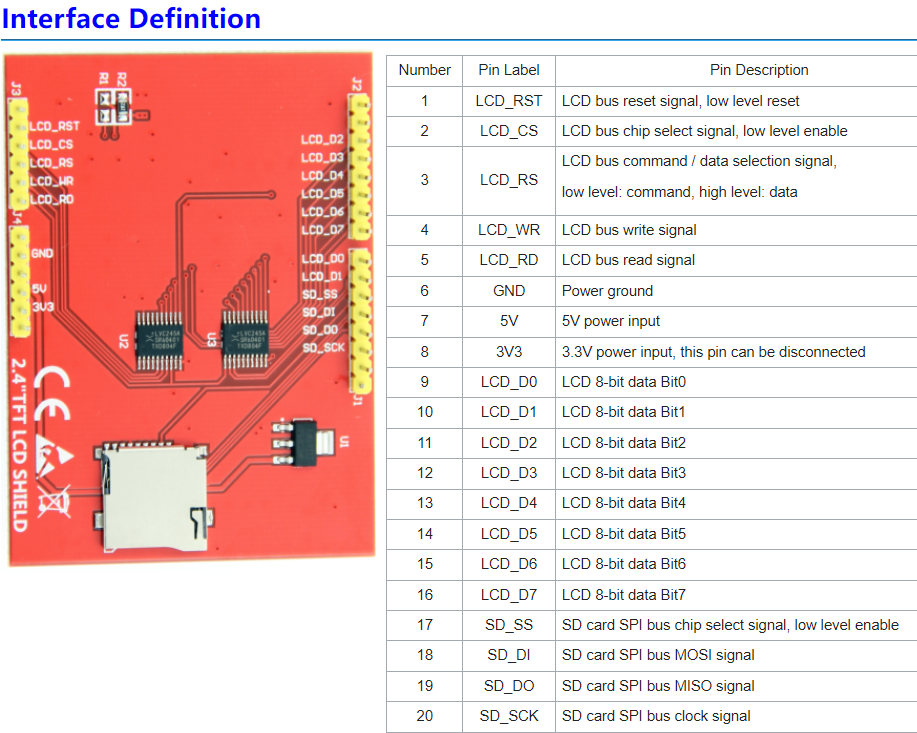
Traditional methods of addressing dyslexia can be tedious for children, hindering their progress. This project aims to create an interactive solution that makes learning enjoyable and effective by leveraging modern technology.

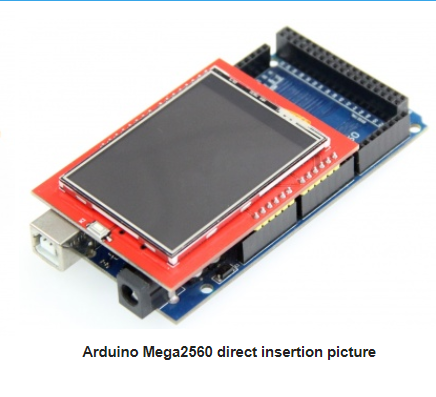
**6. Project Equipment and Components**

6.1 Arduino Overview



The Arduino Mega 2560 is chosen for its ample I/O pins and processing power. With features like ATmega2560 microcontroller and multiple UARTs, it provides the necessary flexibility to handle the TFT display and touch screen interactions seamlessly.

6.2 TFT Screen



The 3.5-inch TFT touch screen display offers a resolution of 320x480 pixels and supports both 16-bit and 8-bit modes. Its integrated touch screen capability enables intuitive user interaction, enhancing the learning experience for children with dyslexia.

7. Theory of Operation

The core of the project is an artificial neural network (ANN) designed to recognize digit patterns drawn by children on the touch screen. The ANN, consisting of input, hidden, and output layers, is trained using a dataset of handwritten digits to predict the digits drawn by children.

**8. Operation**

The system operates through the following steps:

1. Initialization: Arduino Mega initializes the TFT display and touch screen, drawing an 8x8 grid.
2. User Interaction: Children draw digits on the grid using the touch screen.
3. Prediction Trigger: Touching a specific area triggers the neural network to predict the drawn digit.
4. Prediction Display: The predicted digit is displayed on the screen, providing immediate feedback to the child.
5. Repeat: Children can continue drawing and predicting digits, engaging in an interactive learning process.

**9. The Code**

The system's functionality is implemented using Python and C++ programming languages.

9.1 Python Description

Importing Libraries and Modules:

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

from sklearn.datasets import load\_digits

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelBinarizer

numpy: Used for numerical operations on arrays.

tensorflow: The core library for creating and training machine learning models.

Sequential, Dense, Flatten: From tensorflow.keras, these are used to define neural network architectures. Sequential is a linear stack of layers. Dense is a fully connected layer, and Flatten is used to flatten input dimensions.

load\_digits, train\_test\_split, LabelBinarizer: From sklearn, these are used for loading the dataset, splitting it into training and testing sets, and converting labels into a binary (one-hot) format.

Data Loading and Preprocessing:

digits = load\_digits()

data = digits.data / 16.0 # Normalize data

labels = digits.target

The load\_digits() function loads a dataset of handwritten digits. Each data point is an 8x8 image of a digit.

Normalization (data / 16.0) scales the pixel values to the range [0, 1], improving model training efficiency.

Filtering Specific Digits:

mask = np.isin(labels, [0, 1, 2, 3, 4])

data = data[mask]

labels = labels[mask]

This code filters the dataset to include only the digits 0 to 4. This is done using a boolean mask where np.isin checks which elements in labels are in the specified list [0, 1, 2, 3, 4].

Label Encoding:

lb = LabelBinarizer()

labels = lb.fit\_transform(labels)

Converts numeric labels into a one-hot format, which is necessary for training neural networks on classification tasks with more than two classes.

Splitting Dataset:

x\_train, x\_test, y\_train, y\_test = train\_test\_split(data, labels, test\_size=0.2)

Splits the dataset into training and testing sets, with 20% of the data reserved for testing. This helps in evaluating the model on unseen data.

Building the Neural Network:

model = Sequential([

Flatten(input\_shape=(64,)),

Dense(24, activation='relu'), # 24 neurons in the hidden layer

Dense(5, activation='softmax') # Output layer with 5 neurons for digits 0 to 4

])

A neural network is constructed using a sequential model. The Flatten layer converts each 8x8 image into a 64-element vector. The Dense layers are fully connected layers, with ReLU activation for hidden layers and softmax for the output layer to handle multi-class classification.

Compiling and Training the Model:

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=25, validation\_data=(x\_test, y\_test))

The model is compiled with the Adam optimizer and categorical crossentropy loss function, suitable for multi-class classification tasks. It is then trained for 25 epochs on the training data while validating on the test data.

Saving Model Weights:

weights = model.get\_weights()

np.savez('mnist\_weights\_64\_24\_10.npz', \*weights)

After training, the weights and biases of the model are saved in an NPZ file for later use, such as inference or further training.

Importing Libraries:

import numpy as np

numpy is imported again to handle array and numerical operations in this part of the code.

Loading Weights and Biases:

weights\_and\_biases = np.load('mnist\_weights\_64\_24\_10.npz')

This line loads the weights and biases saved from the neural network training in the first cell. The weights are stored in a NumPy archive file ('.npz'), which is a container of multiple arrays saved in a compressed format.

Extracting Individual Arrays:

weights\_input\_hidden = weights\_and\_biases['arr\_0']

bias\_hidden = weights\_and\_biases['arr\_1']

weights\_hidden\_output = weights\_and\_biases['arr\_2']

bias\_output = weights\_and\_biases['arr\_3']

The loaded .npz file contains several arrays, each representing different components of the model's weights and biases. These lines extract each array using keys that reflect their order in the file. The arrays represent weights from input to hidden layer (weights\_input\_hidden), biases for the hidden layer (bias\_hidden), weights from hidden to output layer (weights\_hidden\_output), and biases for the output layer (bias\_output).

Function to Format Arrays:

def format\_array(array, array\_name):

if array.ndim == 1:

array\_str = ", ".join(f"{v:.6f}" for v in array)

formatted\_array = f"float {array\_name}[] = {{{array\_str}}};\n"

else:

array\_str = ",\n".join(

" {" + ", ".join(f"{v:.6f}" for v in row) + "}" for row in array

)

formatted\_array = f"float {array\_name}[][] = {{\n{array\_str}\n}};\n"

return formatted\_array

This function takes an array and its name as input, formats the array values to a fixed number of decimal places, and returns a string representation suitable for use in programming languages like C or C++, where arrays need to be declared with type and size. It distinguishes between 1D and 2D arrays, formatting each accordingly.

Formatting and Printing Arrays:

weights\_input\_hidden = format\_array(weights\_input\_hidden, "weights\_input\_hidden[input\_neurons][hidden\_neurons]")

bias\_hidden = format\_array(bias\_hidden, "bias\_hidden[hidden\_neurons]")

weights\_hidden\_output = format\_array(weights\_hidden\_output, "weights\_hidden\_output[hidden\_neurons][output\_neurons]")

formatted\_bias\_output = format\_array(bias\_output, "bias\_output[output\_neurons]")

print(weights\_input\_hidden)

print(bias\_hidden)

print(weights\_hidden\_output)

print(formatted\_bias\_output)

The weights and biases arrays are formatted using the previously defined function. Each formatted array is then printed. The names passed to the formatting function hint at the structure of the neural network, such as the number of neurons in different layers, making the output easier to understand and use in other contexts, especially when integrating with systems that require explicit type definitions.

9.2 C++ Description

Including Libraries:

#include <Adafruit\_GFX.h>

#include <MCUFRIEND\_kbv.h>

#include <TouchScreen.h>

**Adafruit\_GFX: A library for rendering graphical elements on various displays.**

**MCUFRIEND\_kbv: A library specific for controlling MCUfriend TFT displays.**

**TouchScreen: A library for handling resistive touchscreen functionality.**

**Global Constants and Variables:**

const int MAX = 8;

const int input\_neurons = 64;

const int hidden\_neurons = 24;

const int output\_neurons = 5;

float weights\_input\_hidden[64][24];

float bias\_hidden[];

float weights\_hidden\_output[24][5];

float bias\_output[];

**Defines constants for the neural network architecture, such as the number of neurons in each layer.**

**Arrays for storing weights and biases of the neural network are declared but need to be populated with actual values.**

**TFT Display and Touchscreen Setup:**

MCUFRIEND\_kbv tft;

#define LCD\_CS A3

#define LCD\_CD A2

#define LCD\_WR A1

#define LCD\_RD A0

#define LCD\_RESET A4

#define BLACK 0x0000

#define WHITE 0xFFFF

#define YP A1

#define XM A2

#define YM 7

#define XP 6

#define TS\_MINX 141

#define TS\_MAXX 943

#define TS\_MINY 165

#define TS\_MAXY 835

TouchScreen ts = TouchScreen(XP, YP, XM, YM, 300);

**Configuration for the TFT display and touchscreen, including pin assignments and calibration values for accurate touch input.**

**Setup and Loop Functions:**

void setup() {

// Setup code here

}

void loop() {

// Main program execution loop

}

**setup(): Initializes the serial communication, display settings, and draws the initial UI.**

**loop(): Continuously checks for touch input, updates the display, and handles button interactions and neural network predictions.**

**Neural Network Prediction Implementation:**

void performANNPrediction() {

// Handles digit prediction based on neural network

}

float relu(float x) {

// ReLU activation function

}

void softmax(float input[], float output[], int length) {

// Softmax function for output layer

}

void forward(const uint8\_t input[], float hidden[], float output[]) {

// Forward propagation through the neural network

}

**The neural network functions include activation functions (relu, softmax) and the forward function that computes the output from the input data through the network layers.**

**User Interface Management:**

void drawGrid() {

// Draws grid lines on the display

}

void fillCell(int row, int col, uint16\_t color) {

// Fills a specific cell in the grid based on touch input

}

**Functions for drawing and updating the grid on the TFT display, which represent the input area for digit drawings.**