Jorge L. Alberto
4-27-2024
71993341

EE Design 1 Technical Report
Final Project
PiPico ANN

Introduction

This system will implement an artificial neural network (ANN) that can be trained and tested on the Pi Pico. The goal of the neural network will be to correctly classify whether the number being fed into the network is a zero or one. The drawing of this number will be fed in via a csv file with rows containing the pixel values of an 8x8 pixel image. There will then be systems outside the microcontroller that influence and respond to the results of the neural network.

Systems outside the microcontroller are comprised of a Liquid Crystal Display (LCD) displaying training and testing statuses, analog inputs and outputs, digital inputs and outputs, and a voltage regulator controlling a 9V battery. The analog input allows for a potentiometer to voltage scale analog output. The analog output will go to a speaker that outputs sinusoidal waves. The add something extra requirement will be satisfied with the second digital input to allow for user selection of neural network testing and training. Digital outputs are comprised of three different colored Light Emitting Diodes (LEDs) which correspond to different training and or testing statuses of the neural network.

Methods

Device Physics

LEDs

LEDs are a type of diode that leverages electro luminance to display light of a certain color. Like typical diodes, light emitting diodes are composed of a p-type doped region and an n-type doped region created by adding impurities to the base semiconductor crystal. In between the doped regions is an area known as the depletion region which forms because of hole (from p-type) and free electron (from n-type) recombination.

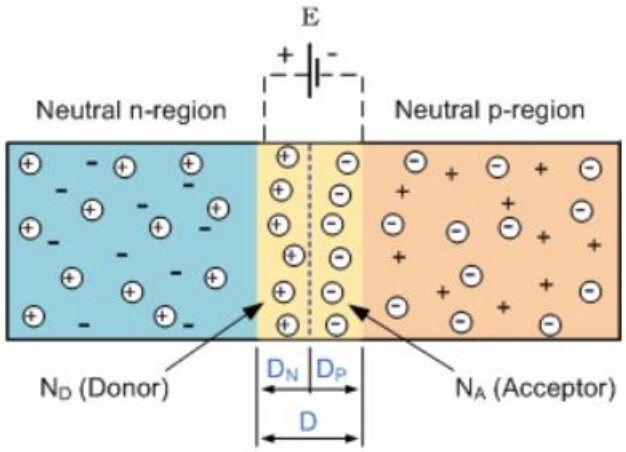

Figure 1: Unbiased PN Junction [1]

In terms of energy, the depletion region represents a difference in energy levels between the n-type and p-type regions. This energy level difference comes from the differences in applied impurities. For p-type materials the applied impurities are trivalent atoms (atom contains three valence electrons), whereas for n-type materials the impurities are pentavalent atoms (atom contains five valence electrons). Trivalent impurities exert lower forces than pentavalent impurities on the outer shell electrons, making the electron orbits slightly larger and hence contain greater energy [3].

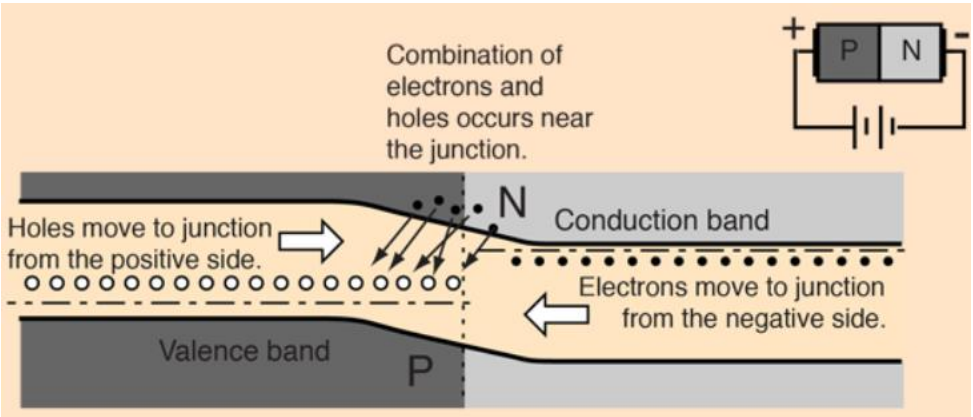

Figure 2: Energy Diagram of Forward Biased PN Junction [2]

The difference between typical diodes and light emitting diodes arises from the energy present in the depletion region. If there is enough energy present in the depletion region, the released energy from the combination of holes and electrons corresponds to a wavelength on the visible spectrum by the formula below.

$$\lambda = \frac{h \times c}{\mathrm{E}}$$

To attain different amounts of energy, different semiconductor materials are required. The different color LEDs also have different forward voltages because of the different materials being used. Below is a table of common semiconductor materials for the three through-hole LED colors being used in this lab.

Color	Wavelength Range (nm)	Forward Voltage (V)	Semiconductor Material
Red	610-760	1.6-2.0	GaAsP
Yellow	570-590	2.1-2.2	GaAsP:N
Green	500-570	1.9-4.0	AlGaP

Table 1: LED Colors and their Corresponding Characteristics

Potentiometer

Unlike LEDs, potentiometers are a passive electrical component. They vary resistance by sliding the position of a contact (wiper) along a uniform resistance. There are two main types of DC potentiometers, rotary and linear potentiometers, in this lab rotary potentiometers were used. Rotary potentiometers are composed of a mechanical rotational mechanism, unlike their linear counterparts which are sensors containing a sliding mechanism. There are several subtypes of rotary potentiometers, the one employed in the lab is a single-turn linear rotary potentiometer which can be rotated to approximately 270°of a full turn. Single-turn rotary potentiometers are commonly used in areas such as volume control (although the logarithmic variation may be more common), where a single turn provides sufficient resolution [4,5].

To further explore the inner working of a potentiometer we can observe the formula for resistance:

$$R_{out} = \frac{\rho \times L}{A}$$

Note that if all parameters but length are kept constant, then resistance is proportional to length. This means that if the potentiometer wiper is adjusted so that it is near ground, then the length between the two terminals is minimized, minimizing the resistance, and minimizing the output voltage [6]. Revealing that potentiometers function as a voltage divider:

$$V_{out} = V_{in} \times \frac{R_{out}}{R_{out} + R}$$

It is important to reemphasize that R_{out} is the resistance between the wiper and ground, and R is the resistance between the wiper and VCC.

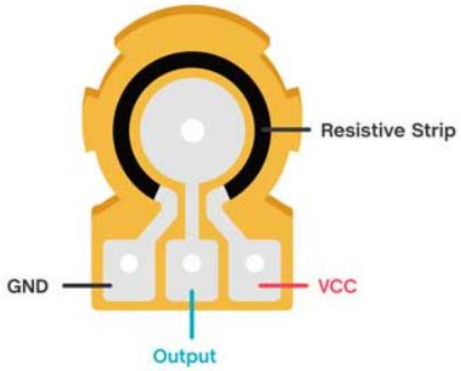

Figure 3: Single Turn Rotary Potentiometer [6]

Common device materials used for the potentiometer's resistive strip are the ones outlined below:

Material	Properties
Carbon Composition	• Common material, Low cost, Reasonable noise and wear
Wire Wound	• Handle high power, Long lasting, & Precise • Limited resolution & Rough feel
Conductive Plastic	• Very smooth feel & High resolution • Can only handle limited power & Expensive
Cermet	• Very stable & Handles high temperatures well • Expensive & Limited lifespan

Table 2: Resistance Strip Materials and Properties [7]

The potentiometers used in this design contain a resistance strip material of cermet [8].

Software

The overall software block diagram is shown below. It starts off with module initialization and loading data, after which a message is displayed to the LCD. After this first sequence of events a while true loop is entered. In this while true loop the user is asked to train or test the ANN by selecting the corresponding button (attached through one of the GPIO pins of the microcontroller). Depending on the selected course of action the code calls on the ANN module functions for training or testing, as well as undergoing interaction with peripherals including displaying an appropriately descriptive LED color and output sound wave. If the user enters the testing phase an additional decision is made by the result of ANN testing. This decision serves the purpose of once again manipulating the peripheral for appropriately descriptive LED color and output sound waves.

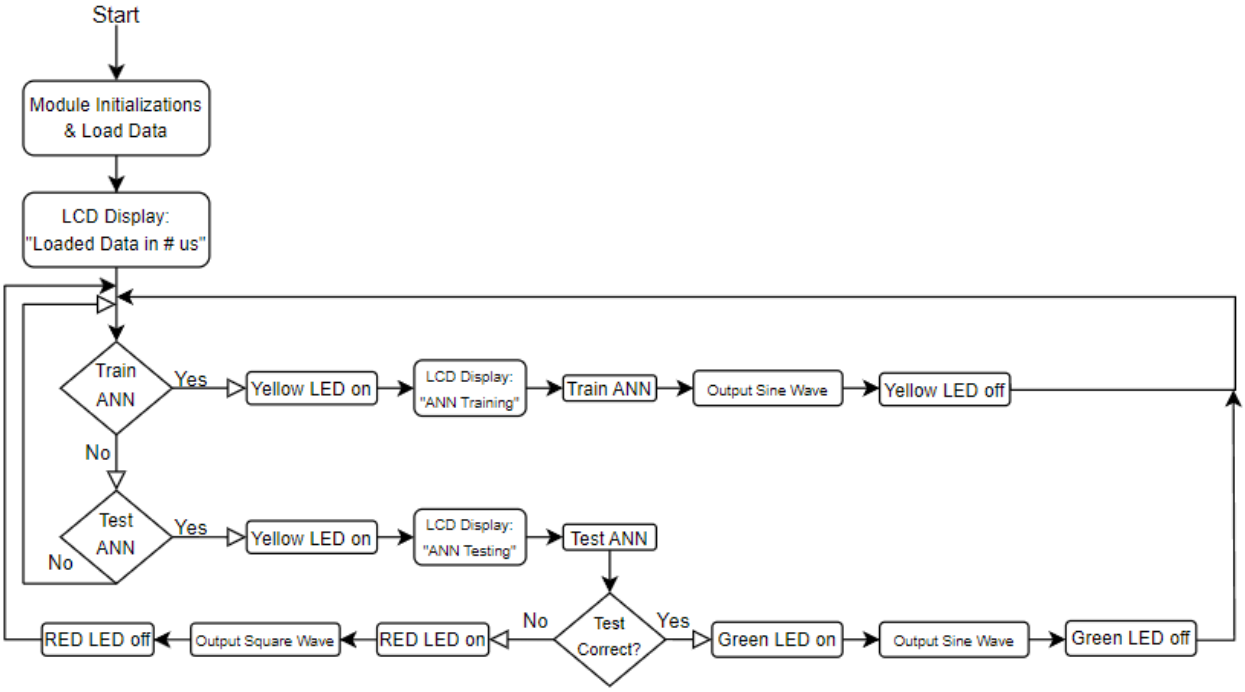

Figure 4: Overall Software Block Diagram

Abstracted away in the software block diagram above are three modules: consts, LCD, and ANN.

Consts and DAC

Consts is the simplest of the three modules, as it only serves as an auxiliary function for selecting different waveforms based on data generated in samples through an online lookup table generator [9]. However, it is important to touch on the reason for selecting these waveforms within our code, which is to output an analog signal based off our digital signal provided by the consts module. In digital signals a good measure of resolution is the number of samples per period. In the code implementation for this project, the number of samples per period was 50, allowing for the digital waveform to be created. However, this digital signal must still be converted to an analog one by passing the elements from the array a consts module function call returns, into the DAC. Because the DAC being used is a 10-bit DAC, the maximum value that an element in the lookup table array can take is 1,023, as 210 - 1 = 1,023 is the total number of levels (the –1 is included to account for 0 being a possible value). DAC's convert the given samples to analog through a filter such as zero order hold, first order hold, or short pulse in the time domain. Then in the frequency domain a lowpass filter is used to remove aliases. Theoretically the aforementioned step would be the last if using the ideal filter (interpolation filter) to convert the samples to analog values. However, using a non-ideal filter is required because the interpolation filter requires an infinite time domain by the nature of it being a sinc function. Therefore, equalization is applied to remove filter distortions and time shifts of the waveform. The DAC accepts data through a 16-bit input word composed of a 4-bit control code, 10-bit input code, and 2-bit don't care, in that order. In order to achieve this word structure an integer representation of the word had to be created to later be processed through the to_bytes function. This was achieved through two left bit shifts, one of size 12 for the control code outside of the while loop, and one of size two for the input code at each sample in lookup table array. Hence the name don't care, the don't care values are not accounted for because they do not impact DAC operation. The two shifted codes were then bitwise or 'ed to achieve the input word. Lastly a separation of the bits into lower and higher bits occurs through the to_bytes function which takes in an integer and outputs a byte array that is passed into the SPI class instance's write function.

LCD

Using Dieter's driver displaying to the LCD becomes as simple as importing the MicroPython driver file, creating an instance of the LCD class (LCD object), and operating on that LCD object through the lcd variable. However, it is important to understand some of the inner workings of the driver code, namely bit banging. Bit banging is a method of communicating over devices that leverages software instead of hardware like

I2C and SPI. This is because in bit banging the microcontroller directly controls timing and state transitions of GPIO pins.

In the “Module Initializations & Load Data” block of the overall software block diagram, the instantiation of the LCD driver will cause entry into the __init__ function which is a Python standard for object construction. This function is not to be confused with init which is a function used to initialize the LCD for communication. The function of most importance when displaying information to the LCD is the LCD’s write function. To reach the write function one must call lcd.print(), which then triggers the _putch function and lastly the write function. Let’s take a look at the write function:

```
def write(self, command, mode):
    """Sends data to the LCD module. """

    # determine if writing a command or data
    data = command if mode == 0 else ord(command)

    # need upper nibble for first loop. lower nibble can use data directly.
    upper = data >> 4

    # write the upper nibble
    for index in range(4):
        bit = upper & 1
        self.data_bus[index].value(bit)
        upper = upper >> 1

    # strobe the LCD, sending the nibble
    self.reg_select_pin.value(mode)
    self.strobe()

    # write the lower nibble
    for index in range(4):
        bit = data & 1
        self.data_bus[index].value(bit)
        data = data >> 1

    # Strobe the LCD, sending the nibble
    self.reg_select_pin.value(mode)
    self.strobe()
    utime.sleep_ms(1)
    self.reg_select_pin.value(1)
```

Figure 5: Dieter’s Write Function Code

The first line of code within the write function determines whether the write function will write data (such as training accuracy) or a command (such as where in the LCD display to navigate to) to the LCD. When called by the LCD’s print function, data will always be written. The data is then separated into lower and upper nibbles through bit shifting. Where each nibble is of size four because there are a total of four buses in the Pico. Strobing the LCD is required after sending each nibble in order to alert the LCD that data is being sent and is ready to be displayed.

ANN

The ANN module was based off the implementation of a simple neural network implementation in Kaggle [10]. Two ANN module functions are called in main: “gradient_descent” for training ANN and “rand_test_prediction” for testing the ANN.

```
def gradient_descent(self, X, Y, alpha, iterations):
    # ANN inits
    W1, b1, W2, b2 = self.init_params()
    # DAC inits
    waveform = waves.get("square")
    control_code = 0b1010 << 12 # 1010 Load DAC B
    j=0
    for i in range(iterations):
        Z1, A1, Z2, A2 = self.forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = self.backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = self.update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            predictions = self.get_predictions(A2)
            acc = self.get_accuracy(predictions, Y)
            # LCD #
            lcd.clear()
            lcd.print(f'Epoch {i}')
            lcd.go_to(0,1)
            lcd.print(f'{(acc*100):.{2}f}% accuracy')
            # DAC #
            x = 250
            input_code = x << 2
            data = control_code | input_code
            try:
                cs(0)
                buf = data.to_bytes(2,'big')
                spi_0.write(buf)
            finally:
                    cs(1)
            j+=1

    return W1, b1, W2, b2
```

Figure 6: Training Function in ANN Module

Gradient descent is a fundamental component of neural networks that is composed of four steps: initialization, forward propagation, backpropagation, and updating.

```
def init_params(self):
    W1 = [ [.5] * (self.n-1) for digit in range(self.num_labels)]
    b1 = [ [.21] * 1 for digit in range(self.num_labels)]
    W2 = [ [.32] * self.num_labels for digit in range(self.num_labels)]
    b2 = [ [0.0] * 1 for digit in range(self.num_labels)]

    W1 = np.array(W1)
    b1 = np.array(b1)
    W2 = np.array(W2)
    b2 = np.array(b2)

    return W1, b1, W2, b2
```

Figure 7: Parameter Initialization Function in ANN Module

Initialization is simply the initialization of the weights in the neural network. It is typically random, but I have selected arbitrary deterministic values for reproducibility.

```
def forward_prop(self, W1, b1, W2, b2, X):
    Z1 = np.dot(W1, X) + b1
    A1 = self.ReLU(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = self.softmax(Z2)
    return Z1, A1, Z2, A2
```

Figure 8: Forward Propagation Function in ANN Module

Forward propagation is the calculation of the neural network result through matrix multiplication of the input matrix represented in the variable “X.” The W’s and b’s shown in the function are simply the neural network parameters, which are the weights and biases of layers one and two.

```
def backward_prop(self, Z1, A1, Z2, A2, W1, W2, X, Y):
    dZ2 = A2 - self.one_hot_Y_train
    dW2 = 1 / self.m * np.dot(dZ2, A1.T)
    db2 = 1 / self.m * np.sum(dZ2)
    dZ1 = np.dot(W2.T, dZ2) * self.ReLU_deriv(Z1)
    dW1 = 1 / self.m * np.dot(dZ1, X.T)
    db1 = 1 / self.m * np.sum(dZ1)
    return dW1, db1, dW2, db2
```

Figure 9: Backwards Propagation Function in ANN Module

Backward propagation is the calculation of the neural network error. First the error in the last layer is computed by comparing the actual output present in "self.one_hot_Y_train" to the predicted output in "A2."Then, gradients of the loss function with respect to the network parameters are computed using the chain rule of calculus. In more concrete terms, the error matrix is dot product multiplied by the network parameters to measure the degree to which each network parameter contributes to the error. The "1/m" term simply averages the result over the number of neurons in the layer. This process continues going from the output layer to the input, hence the name backward in backward propagation.

```
def update_params(self, W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 -= alpha * dW1
    b1 -= alpha * db1
    W2 -= alpha * dW2
    b2 -= alpha * db2
    return W1, b1, W2, b2
```

Figure 10: Parameter Update in ANN Module

Lastly, the contributed error from each neuron in the neural network is subtracted from the parameters. The subtraction of the contributed error can be increased by increasing alpha (also known as the learning rate). While this might allow for the ANN to train faster, it could also lead to the ANN being unable to attain sufficient accuracy because it is descending a gradient too far in one direction.

After all these functions are called, one epoch is completed. Then LCD functions are called to update the user and another epoch starts, once again calling the same four functions to minimize ANN error.

```
def rand_test_prediction(self, W1, b1, W2, b2):
    index = random.randrange(0, len(self.X_train.T), 1)
    current_image = np.array([self.X_train.T[index]]).T
    prediction = self.make_predictions(W1, b1, W2, b2, current_image)
    label = self.Y_train[index]
    # LCD #
    lcd.clear()
    lcd.print(f'Expected {prediction}')
    lcd.go_to(0,1)
    lcd.print(f'Predicted {label}     ')
    return prediction == label
```

Figure 11: Testing Function in ANN Module

When the user would like to test the ANN against a random sample from the dataset the ANN was not trained on, then the above function is called. The random sample is selected from the random generation of an index, which is then used to access the dataset at that index. The make predictions functions calls the previously discussed forward propagation function and returns the predicted value of 0 or 1. The function returns a Boolean value based off the predicted value was the same as the label. Corresponding LCD actions are also taken to alert the user.

Hardware

The overall hardware block diagram is shown below. At its core is the power provided by the nine-volt battery and power regulation system. Inputs are taken in from a potentiometer for the analog input, and two buttons for the digital inputs. The analog input controls the strength of the output analog waveform through the speaker, and the digital inputs control selection of testing or training states. Depending on the selected course of action the code calls on the ANN module functions for training or testing. This then leads to interaction with peripherals to light up one of the LEDs, output a waveform through the eight-ohm speaker, and display information on the LCD. LEDs and potentiometers were covered in the Device Physics section, and the DAC was covered alongside the Consts module because they are so heavily linked together, therefore I will focus on the other hardware components that require functionality.

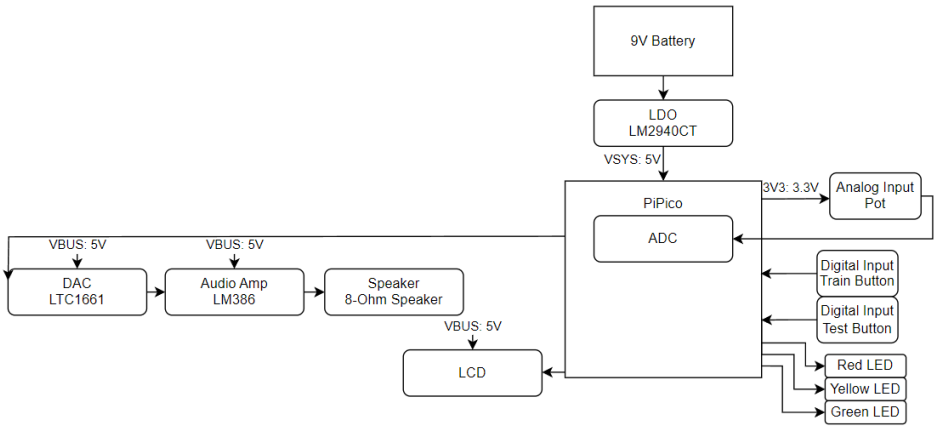

Figure 12: Hardware Block Diagram

Power Regulation

The LDO offers a low dropout voltage, so that the minimum voltage needed to be supplied over the output for the circuit to function is relatively small. The LDO also offers smooth output waveforms and voltage regulation across various output currents. LDO's are often used for wireless communications, portable equipment, industrial and automotive applications, and digital core supplies. LDO's are needed for wireless communications because a clean power supply is important for sensitive radio frequency (RF) circuits. For portable equipment their ability to be manufactured in a small size is crucial. In industrial and automotive applications the fact that LDO's do not generate electromagnetic interference (EMI), unlike switching regulators, ensures AM frequencies are better received. Digital cores are the main processing unit of a digital system and utilize higher power LDO's for computational power, as well as for their ability to improve power efficiency [11].

Audio Amplifier

After receiving the analog output from the DAC, it must be amplified before outputting to the speaker. To do this we can use an audio amplifier, which hence the name, is specifically designed for audio applications, the LM386 is one such circuit. It has a default gain of $20\frac{V}{V}$ and implements the internal circuitry required to amplify audio (active high pass filter with appropriate resistance and capacitance values). Components attached to the LM386 in the final design, as seen in the Altium Schematic, are capacitors to remove DC offsets, a bypass capacitor, and the Zobel network. Zobel networks help with impedance matching for inductive loads such as an 8Ω speaker. The output capacitor removes the DC component present with the reference ground created by a bias voltage. And bypass capacitors help keep stable voltage levels during transients.

PCB Design

To implement all these components physically, I used Altium to create the files required to be sent to the manufacturer for PCB construction. Below is the realization of all the components in Altium.

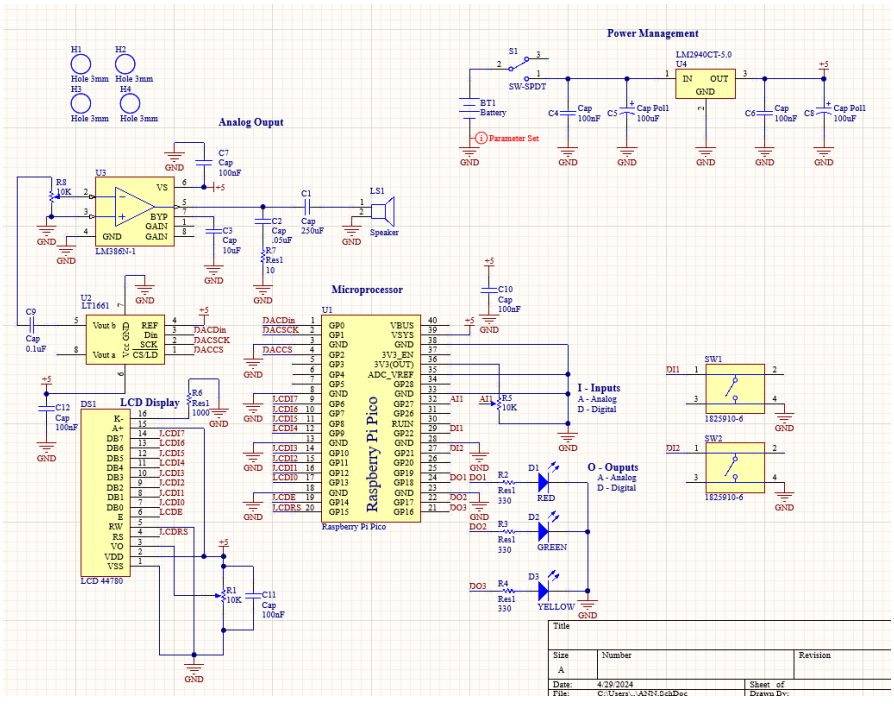

Figure 13: Altium Schematic

The schematic is the first part of the realization process, in which I took my hardware block diagram along with documentation to correctly net components. It is clear to see that several bypass capacitors have been placed. This is mainly for bypass capacitor's

ability to help mitigate power fluctuation. This ensures near constant voltages are seen at power and ground pins on integrated circuits (ICs). Bypass capacitors can also ensure stable and clean signals at various frequencies by filtering out high and low frequency noises. Larger polarized capacitors are typically used for filtering out low frequencies because they charge and discharge slower. Smaller ceramic capacitors can charge and discharge faster, therefore filtering out high frequencies. Bypass capacitors should be placed closely to the power pins of the integrated circuit (IC) to minimize encountered parasitic inductances.

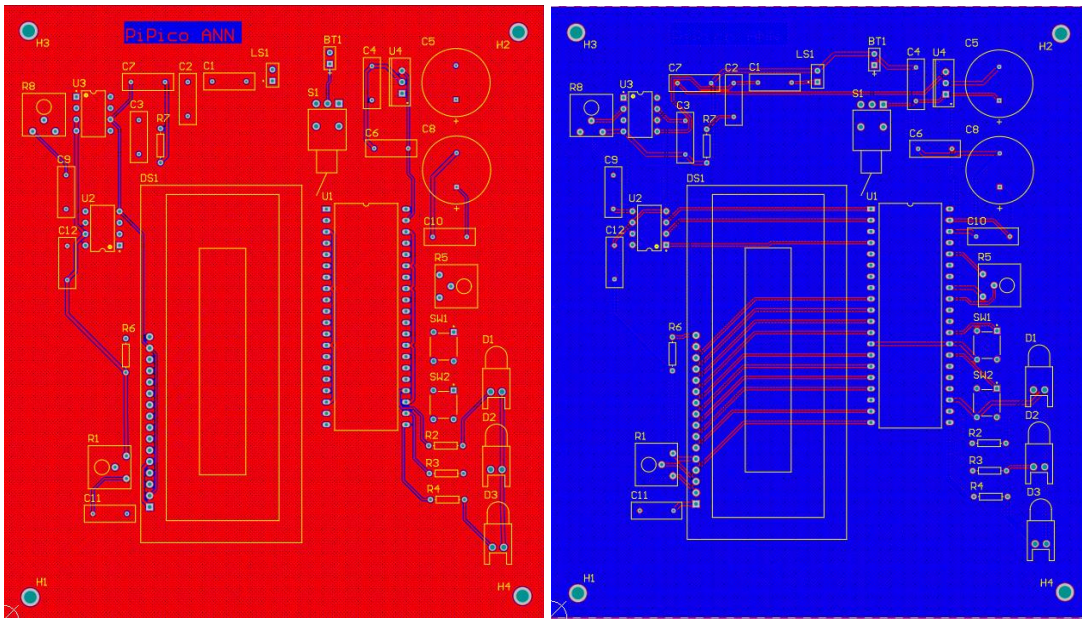

Figure 14: Altium Board Layout

The top layer of the PCB is shown to the left, it is connected to +9V. To the right is the bottom layer of the PCB which is connected to ground. These connections make routing traces much simpler and reduce electromagnetic interference.

Bill of Parts

For reproducibility and considerations of cost constraints, I have attached a bill of parts.

Part Number	Description	Designator	Supplier Unit Price	Quantity
Battery	Multicell Battery	BT1	$1.00	1
Cap	Capacitor	C1, C2, C3, C4, C6, C7, C9, C10, C11, C12	$1.17	10
Cap Pol1	Polarized Capacitor (Radial)	C5, C8	$1.40	2
LED0	Typical INFRARED GaAs LED	D1	$0.34	1
LED0	Typical INFRARED GaAs LED	D2	$0.34	1
LED0	Typical INFRARED GaAs LED	D3	$0.34	1
LCD 44780		DS1	$2.80	1
Speaker	Loudspeaker	LS1	$1.99	1
RPot_5	Potentiometer	R1, R5, R8	$3.15	3
Res1	Resistor	R2, R3, R4, R6, R7	$0.50	5
SW-SPDT	SPDT Subminiature Toggle Switch, Right Angle Mounting, Vertical Actuation	S1	$1.75	1
CMP-1684-00021	Tact Switch, SPST-NO, 0.05 A, -35 to 85 degC, 4-Pin THD, RoHS, Bulk	SW1, SW2	$0.20	2
Raspberry Pi Pico		U1	$4.00	1
LT1661		U2	$6.24	1
LM386N-1	Low-Voltage Audio Power Amplifier	U3	$1.28	1
LM2940CT-5.0_1	1A Low Dropout Regulator, 3-pin TO-220	U4	$2.62	1
			$29.12	33

Table 3: Bill of Parts

Results

The components described in the previous sections resulted in the final project described by the elements shown below.

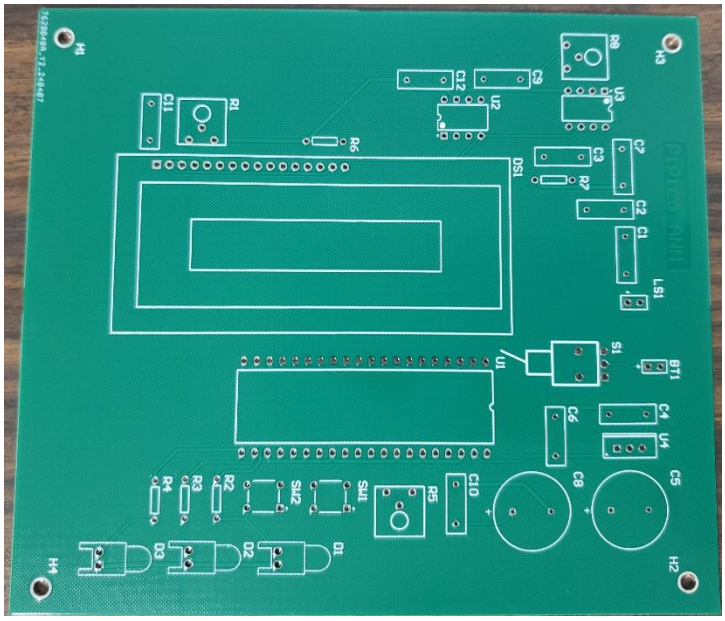

Figure 15: Final PCB

Moving the design from my breadboard to my PCB was straightforward because I designed my breadboard based off my PCB. This meant all software pinouts were initialized correctly, and any errors had been debugged.

Figure 16: LCD Display After Loading Data

The realization of moving the components from my breadboard to my PCB is shown in this above image. Here the PiPico enters the initial state where it loads the data from the NumPy formatted file. This state is shown in the software block diagram by the block “LCD Display: “Loaded Data in # us”.”

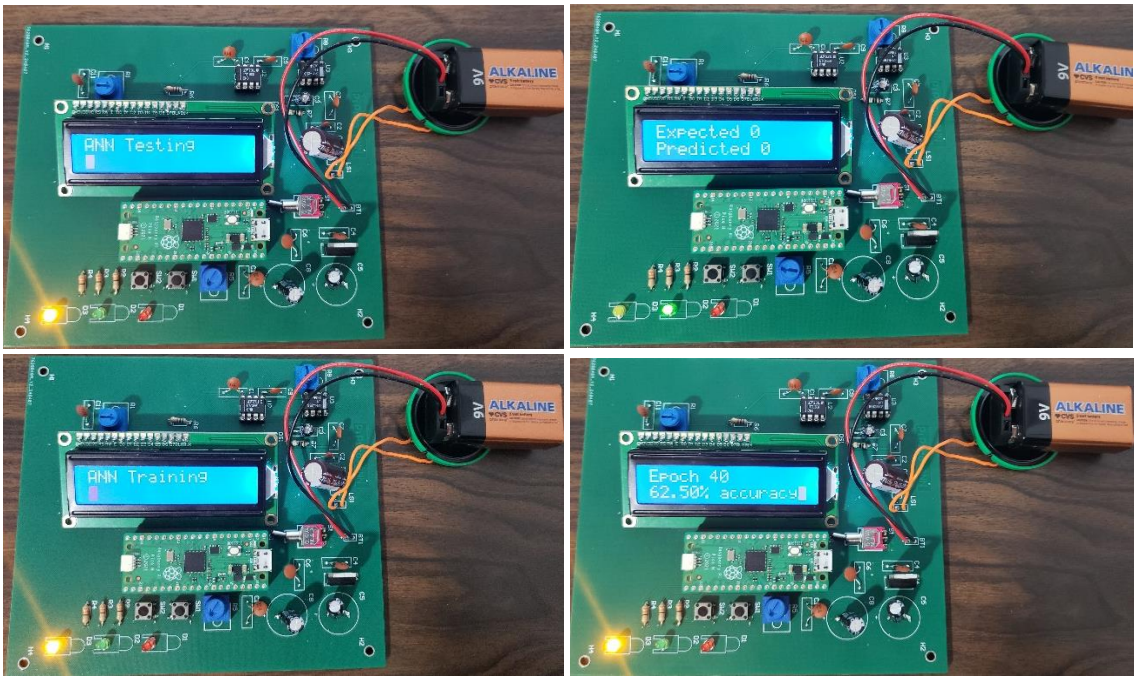

Figure 17: Different LCD Display States Based on User Action

Once the data is loaded the user can select between the testing and training states. The display of LCD when these states are selected is shown on the left and are part of the "LCD Display: "ANN Training"," and "LCD Display: "ANN Testing"" blocks of the software block diagram. Internal logic for each of these states then outputs the corresponding LCD text on the right. This internal logic is represented in the software block diagrams' "Train ANN" and "Test ANN" blocks. For training, the accuracy of the model is output after each ten passes (epochs) through the neural network, until the specified number of epochs (specified in code) is reached. In testing, the LCD display can only differ from the one depicted above by having different values for expected and or predicted values of 0 or 1 (binary classification).

Figure 18: Analog outputs

Pictured above are the analog outputs required for the project, which in my case was a sine wave. On the left is the sine wave at the output of the DAC, it clearly avoids railing, distortion, and is output correctly. However, on the right side the output of the sine wave at the speaker is displayed, and this sine wave is not free of distortion.

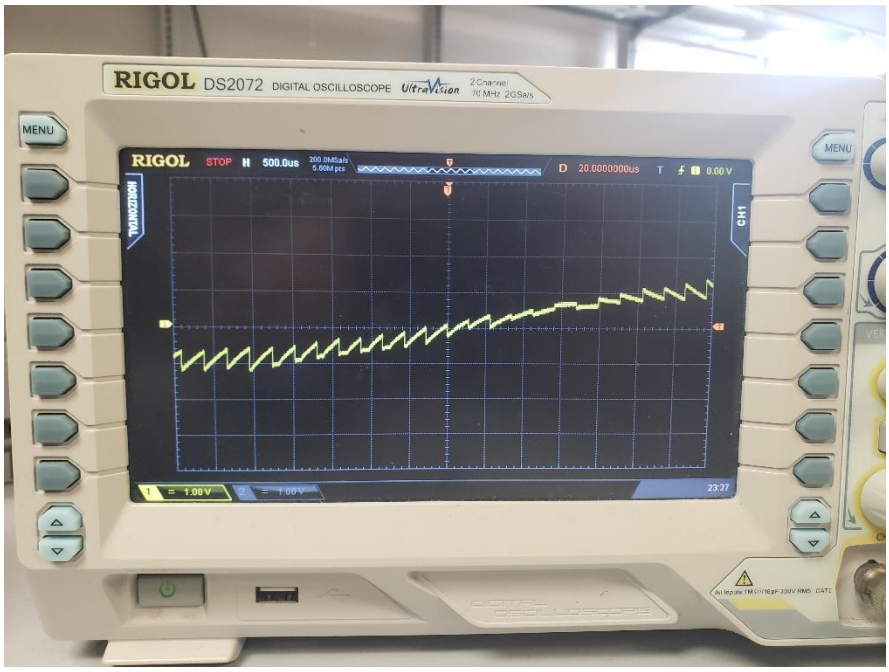

Figure 19: Zoomed in Analog Output at Speaker

Zooming in on the distorted sine wave at the speaker end, we can see that high frequency components are being attenuated. This is evident by the fact that the “steps” the sine wave is supposed to take between voltage levels are not steps, but rather some nature of ramps. These ramps are known to be lower frequency components because they do not instantly reach the next voltage level, meaning that they have a slower transition and hence lower frequency.

Discussion/Conclusion

Several trials were encountered throughout this project. Initially the main slowing of progress was simply the inertia present to start the project. The ideas and knowledge required to implement everything were present, but steps for realization of my idea were not. To overcome this I started on the hardware block diagram first, allowing me to get a clear picture of what was ahead. This allowed me to easily implement the idea on my breadboard, combining previous modules into a larger systemic project. Some concerns did pop up occasionally during breadboarding, namely my confusion around the ADC_VREF pin on the PiPico. At first, I thought this pin was to be grounded, but then I realized that it functioned like the reference pin on my DAC (in which case it was supposed to be high). Another source of confusion was the non-one to one functionality between micro-Python and Python. There were several times I attained functionality in Python, but not in micro-Python.

All these trials were eventually overcome, and project implementation was successful. Some things that should be improved for future use of the project include improved smoothness of analog output going to speaker. The analog output going to the speaker was choppy because of bypass capacitor values containing capacitance values that were too large. This occurs because at each output sample transition there is a high frequency component caused by the sudden switching from one voltage level to another. That high frequency component is over-attenuated by my selected capacitor values. To address this I would experiment with different capacitance values first in LTSpice and then through breadboarding.

References

[1] Storr, W. (2022, August 6). *PN junction theory for semiconductor diodes*. Basic Electronics Tutorials. https://www.electronics-tutorials.ws/diode/diode_2.html

[2] *P-N Energy Bands*. Biasing of P-N Junctions. (n.d.). http://hyperphysics.phy-astr.gsu.edu/hbase/Solids/pnjun2.html

[3] *Doped semiconductors*. Unacademy. (2022, April 20). https://unacademy.com/content/upsc/study-material/physics/doped-semiconductors/

[4] Electrical4U. (2020, October 28). *Potentiometer: Definition, types, and working principle*. https://www.electrical4u.com/potentiometer/#google_vignette

[5] *Potentiometers: TTI, Inc.*. Potentiometers | TTI, Inc. (n.d.). https://www.tti.com/content/ttiinc/en/resources/blog/potentiometers.html

[6] Jeff Smoot, V. of A. E. and M. C. at C. D. (2023, May 31). *The Complete Guide to Potentiometers*. DigiKey. https://www.digikey.com/en/articles/the-complete-guide-to-potentiometers

[7] *Potentiometer: Resistor types: Resistor guide*. EEPower. (n.d.). https://eepower.com/resistor-guide/resistor-types/potentiometer/#

[8] COM-09806 Sparkfun Electronics | Potentiometers, variable resistors | DigiKey. (n.d.). https://www.digikey.com/en/products/detail/sparkfun-electronics/COM-09806/7319606

[9] Ppelikan (2024) Ppelikan, Dr LUT. Available at: https://ppelikan.github.io/drlut/ (Accessed: 03 March 2024).

[10] Wwsalmon. (2020, November 24). *Simple mnist NN from scratch (numpy, no TF/keras)*. Kaggle. https://www.kaggle.com/code/wwsalmon/simple-mnist-nn-from-scratch-numpy-no-tf-keras

[11] Brokaw, P. (2022) Low-dropout linear regulator (LDO) Application Tutorial, Analog Devices. Available at: https://www.analog.com/en/resources/technical-articles/lowdropout-linear-regulator ldo-application-tutorial.html (Accessed: 02 February 2024).

Appendix

Main

```
from drivers.LCD import LCD
from machine import SoftSPI, Pin, ADC
from ulab import numpy as np
from consts import Waves
from ANN import ANN
import utime

"""instantiations"""
lcd = LCD(enable_pin=14,           # Enable Pin, int
          reg_select_pin=15,       # Register Select, int
          data_pins=[9,8,7,6]   # Data Pin numbers for the upper nibble. list[int]
          )

spi_0 = SoftSPI(baudrate=500000,
        polarity=0,
        phase=0,
        bits=8,
        firstbit=SoftSPI.MSB,
        sck=Pin(1),
        mosi=Pin(0),
        miso=machine.Pin(3))
cs = Pin(2, mode=Pin.OUT, value=1)

analog_pin = Pin(27, machine.Pin.IN)
analog_pin = ADC(27)

btn1 = Pin(22, machine.Pin.IN, machine.Pin.PULL_UP) # 0 pressed, 1 not pressed
btn2 = Pin(21, machine.Pin.IN, machine.Pin.PULL_UP) # 0 pressed, 1 not pressed

red_led = Pin(18, machine.Pin.OUT)
green_led = Pin(17, machine.Pin.OUT)
yellow_led = Pin(16, machine.Pin.OUT)

waves = Waves()

start_time = utime.time_ns()
data = []
data = np.load('./data/zerosNones_200.npy')
end_time = utime.time_ns()
data_processing_time = int(end_time - start_time)/1_000_000
ann = ANN(data=data, num_labels=2)
```

```
"""initializations"""
lcd.init()
lcd.clear()

conversion_factor = 1 / 65535.

green_led.value(0)
red_led.value(0)
yellow_led.value(0)

control_code = 0b1010 << 12 # 1010 Load DAC B
waveform = waves.get("sin")

lcd.print(f'Loaded Data in    ')
lcd.go_to(0,1)
lcd.print(f'{data_processing_time} us     ')

"""ML vars"""
W1,b1,W2,b2 = ann.init_params()

def output_wave(waveform_name, freq=500, signal_periods=1_000):
    waveform = waves.get(waveform_name)
    for t in range(signal_periods):
        for i in range(50): # 50 is len of array
            analog_input = analog_pin.read_u16()
            volt_input = analog_input * conversion_factor
            input_code = int(waveform[i]*volt_input) << 2
            data = control_code | input_code
            try:
                cs(0)
                buf = data.to_bytes(2,'big')
                spi_0.write(buf)
            finally:
                cs(1)
            utime.sleep_us(int(1000000/(50*freq))) # 10Hz @ 100ms - 100Hz @ 10ms

while True:
    train = not btn1.value()
    test = not btn2.value()

    ### Train ANN ###
    if train:
        yellow_led.value(1)
        lcd.clear()
```

```
        lcd.print(f'ANN Training    ')
        lcd.go_to(0,1)
        lcd.blink()
        utime.sleep(2)
        W1, b1, W2, b2 = ann.gradient_descent(ann.X_train, ann.Y_train, .8, 250)
        output_wave("sin")
        yellow_led.value(0)

    ### Test ANN ###
    elif test:
        yellow_led.value(1)
        lcd.clear()
        lcd.print(f'ANN Testing     ')
        lcd.go_to(0,1)
        lcd.blink()
        utime.sleep(2)
        yellow_led.value(0)
        test_correct = ann.rand_test_prediction(W1, b1, W2, b2)

        if test_correct:
            green_led.value(1)
            output_wave("sin")
            green_led.value(0)
        elif not test_correct:
            red_led.value(1)
            output_wave("square")
            red_led.value(0)
```

ANN Module

```
from ulab import numpy as np
import time
import random
from consts import Waves
from drivers.LCD import LCD
from machine import SoftSPI, Pin
import machine

lcd = LCD(enable_pin=14,           # Enable Pin, int
          reg_select_pin=15,       # Register Select, int
          data_pins=[9,8,7,6]   # Data Pin numbers for the upper nibble. list[int]
          )
spi_0 = machine.SoftSPI(baudrate=500000,
        polarity=0,
```

```
        phase=0,
        bits=8,
        firstbit=SoftSPI.MSB,
        sck=Pin(1),
        mosi=Pin(0),
        miso=machine.Pin(3))
cs = machine.Pin(2, mode=Pin.OUT, value=1)
waves = Waves()


class ANN:

    def __init__(self, data, num_labels) -> None:
        self.data = data
        self.m, self.n = data.shape
        self.num_labels = num_labels

        data_test = self.data[0:int(self.m*.2)].T
        self.Y_test = data_test[self.n-1]   # target vals, last column
        self.one_hot_Y_test = self.one_hot(self.Y_test)
        self.X_test = data_test[0:self.n-1] / 255. # pixel vals normalized

        data_train = self.data[int(self.m*.2):self.m].T
        self.Y_train = data_train[self.n-1]   # target vals, last column
        self.one_hot_Y_train = self.one_hot(self.Y_train)
        self.X_train = data_train[0:self.n-1] / 255. # pixel vals normalized

    def init_params(self):
        W1 = [ [.5] * (self.n-1) for digit in range(self.num_labels)]
        b1 = [ [.21] * 1 for digit in range(self.num_labels)]
        W2 = [ [.32] * self.num_labels for digit in range(self.num_labels)]
        b2 = [ [0.0] * 1 for digit in range(self.num_labels)]

        W1 = np.array(W1)
        b1 = np.array(b1)
        W2 = np.array(W2)
        b2 = np.array(b2)

        return W1, b1, W2, b2

    def ReLU(self, Z):
        return np.maximum(Z, 0)

    def softmax(self, Z):
        exp = np.exp(Z - np.max(Z))
```

```
        return exp / np.sum(exp, axis=0)

    def forward_prop(self, W1, b1, W2, b2, X):
        Z1 = np.dot(W1, X) + b1
        A1 = self.ReLU(Z1)
        Z2 = np.dot(W2, A1) + b2
        A2 = self.softmax(Z2)
        return Z1, A1, Z2, A2

    def ReLU_deriv(self, Z):
        return Z > 0

    def one_hot(self, Y):
        one_hot_Y = np.zeros((Y.size, self.num_labels))
        for i in range(Y.size):
            one_hot_Y[i][Y[i]] = 1
        one_hot_Y = one_hot_Y.T
        return one_hot_Y

    def backward_prop(self, Z1, A1, Z2, A2, W1, W2, X, Y):
        dZ2 = A2 - self.one_hot_Y_train
        dW2 = 1 / self.m * np.dot(dZ2, A1.T)
        db2 = 1 / self.m * np.sum(dZ2)
        dZ1 = np.dot(W2.T, dZ2) * self.ReLU_deriv(Z1)
        dW1 = 1 / self.m * np.dot(dZ1, X.T)
        db1 = 1 / self.m * np.sum(dZ1)
        return dW1, db1, dW2, db2

    def update_params(self, W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
        W1 -= alpha * dW1
        b1 -= alpha * db1
        W2 -= alpha * dW2
        b2 -= alpha * db2
        return W1, b1, W2, b2

    def get_predictions(self, A2):
        return np.argmax(A2, axis=0)

    def get_accuracy(self, predictions, Y):
        return np.sum(predictions == Y) / Y.size

    def gradient_descent(self, X, Y, alpha, iterations):
        # ANN inits
        W1, b1, W2, b2 = self.init_params()
        # DAC inits
```

```
        waveform = waves.get("square")
        control_code = 0b1010 << 12 # 1010 Load DAC B
        j=0
        for i in range(iterations):
            Z1, A1, Z2, A2 = self.forward_prop(W1, b1, W2, b2, X)
            dW1, db1, dW2, db2 = self.backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
            W1, b1, W2, b2 = self.update_params(W1, b1, W2, b2, dW1, db1, dW2,
db2, alpha)
            if i % 10 == 0:
                predictions = self.get_predictions(A2)
                acc = self.get_accuracy(predictions, Y)
                # LCD #
                lcd.clear()
                lcd.print(f'Epoch {i}')
                lcd.go_to(0,1)
                lcd.print(f'{(acc*100):.{2}f}% accuracy')
                # DAC #
                x = 250
                input_code = x << 2
                data = control_code | input_code
                try:
                    cs(0)
                    buf = data.to_bytes(2,'big')
                    spi_0.write(buf)
                finally:
                        cs(1)
                j+=1

        return W1, b1, W2, b2

    def make_predictions(self, W1, b1, W2, b2, X):
        _, _, _, A2 = self.forward_prop(W1, b1, W2, b2, X)
        return self.get_predictions(A2)

    def test_prediction(self, index, W1, b1, W2, b2):
        current_image = X_train[:, index, None]
        prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)
        label = Y_train[index]
        print("Prediction: ", prediction)
        print("Label: ", label)

    def rand_test_prediction(self, W1, b1, W2, b2):
        index = random.randrange(0, len(self.X_train.T), 1)
        current_image = np.array([self.X_train.T[index]]).T
        prediction = self.make_predictions(W1, b1, W2, b2, current_image)
```

```
        label = self.Y_train[index]
        # LCD #
        lcd.clear()
        lcd.print(f'Expected {prediction}')
        lcd.go_to(0,1)
        lcd.print(f'Predicted {label}     ')
        return prediction == label

if __name__ == '__main__':
    data_list = []
    # Start timing
    start_time = time.time()

    data = np.load('./data/zerosNones_200.npy')

    # Stop timing
    end_time = time.time()

    print(f"Processed Data in {end_time - start_time} seconds")

    ann = ANN(data=data, num_labels=2)

    W1, b1, W2, b2 = ann.gradient_descent(ann.X_train, ann.Y_train, .4, 1_000)
```

Consts Module

```
from machine import Pin
import utime


class Waves:
    def __init__(self) -> None:
        # Formula: sin(2*pi*t/T)
        self.sin = [512,   576,   639,   700,   759,   813,   862,
                    907,   944,   975,   999,  1015,  1023,  1023,
                   1015,   999,   975,   944,   907,   862,   813,
                    759,   700,   639,   576,   512,   448,   385,
                    324,   265,   211,   162,   117,    80,    49,
                     25,     9,     1,     1,     9,    25,    49,
                     80,   117,   162,   211,   265,   324,   385,
                    448]
        # Formula: step(t%T-T*1/4)-step(t%T-T*3/4)
        self.square = [0,     0,     0,     0,     0,     0,     0,
                       0,     0,     0,     0,     0,     0,  1023,
```

```
                1023,  1023,  1023,  1023,  1023,  1023,  1023,
                1023,  1023,  1023,  1023,  1023,  1023,  1023,
                1023,  1023,  1023,  1023,  1023,  1023,  1023,
                1023,  1023,  1023,     0,     0,     0,     0,
                   0,     0,     0,     0,     0,     0,     0,
                   0 ]
        # Formula: 2*(t%T)/T-1
        self.saw = [0,    20,    41,    61,    82,   102,   123,
                    143,   164,   184,   205,   225,   246,   266,
                    287,   307,   328,   348,   369,   389,   410,
                    430,   451,   471,   492,   512,   532,   553,
                    573,   594,   614,   635,   655,   676,   696,
                    717,   737,   758,   778,   799,   819,   840,
                    860,   881,   901,   922,   942,   963,   983,
                   1004]
        # Formula: 4*abs(t/T-floor(t/T+1/2))-1
        self.triangle = [0,    41,    82,   123,   164,   205,   246,
                    287,   328,   369,   410,   451,   492,   532,
                    573,   614,   655,   696,   737,   778,   819,
                    860,   901,   942,   983,  1023,   983,   942,
                    901,   860,   819,   778,   737,   696,   655,
                    614,   573,   532,   492,   451,   410,   369,
                    328,   287,   246,   205,   164,   123,    82,
                     41]

        self.small = [0,    41,    82,   12,   14,   20,   24,
           287,   328,   369,   41,   45,   49,   53,
           57,   61,   65,   69,   73,   77,   81,
           86,   901,   942,   983,  1023,   93,   942,
           91,   80,   89,   78,   77,   69,   65,
           64,   53,   52,   42,   41,   41,   36,
           32,   28,   24,   25,   16,   13,    82,
            41]

    def get(self, wave):
        if wave == "sin":
            return self.sin
        elif wave == "square":
            return self.square
        elif wave == "saw":
            return self.saw
        elif wave == "triangle":
            return self.triangle
        elif wave == "small":
            return self.small
```

LCD Module

```
# ----------------------------------------
#                 NOTES
# ----------------------------------------
"""
Dieter Steinhauser
5/2023
Parallel LCD class, Configured for 4-bit communication.
"""

# ----------------------------------------
#               IMPORTS
# ----------------------------------------

from machine import Pin
import utime

# ----------------------------------------
#            LCD Instantiation
# ----------------------------------------

# add this code to the beginning of your main file to instantiate the LCD

# GPIO
# lcd = LCD(enable_pin=0,           # Enable Pin, int
        # reg_select_pin=1,        # Register Select, int
        # data_pins=[2, 3, 4, 5]   # Data Pin numbers for the upper nibble.
list[int]
        # )

# lcd.init()
# lcd.clear()
# lcd.cursor_on()
# lcd.blink()


# ----------------------------------------
#                 LCD Class:
# ----------------------------------------


class LCD:
```

```
    """The LCD class is meant to abstract the LCD driver further and streamline
development."""


    def __init__(self, enable_pin: int, reg_select_pin: int, data_pins: list) ->
None:
        """Object initialization"""

        self.enable_pin = Pin(enable_pin, Pin.OUT)
        self.reg_select_pin = Pin(reg_select_pin, Pin.OUT)
        self._data_pins = data_pins
        self.data_bus = []

        # Configure the pins of the device.
        self._configure()
        utime.sleep_ms(120)

    # ----------------------------------------

    def _configure(self):
        """Creates the data bus object from the pin list. """

        # Configure the pins of the device.
        for element in self._data_pins:
            self.data_bus.append(Pin(element, Pin.OUT))

    # ----------------------------------------

    def init(self):
        """Initializes the LCD for communication."""

        # clear values on data bus.
        for index in range(4):
            self.data_bus[index].value(0)
        utime.sleep_ms(50)

        # initialization sequence.
        self.data_bus[0].value(1)
        self.data_bus[1].value(1)
        self.strobe()
        utime.sleep_ms(10)

        self.strobe()
        utime.sleep_ms(10)
```

```
        self.strobe()
        utime.sleep_ms(10)

        self.data_bus[0].value(0)
        self.strobe()
        utime.sleep_ms(5)

        self.write(0x28, 0)
        utime.sleep_ms(1)

        self.write(0x08, 0)
        utime.sleep_ms(1)

        self.write(0x01, 0)
        utime.sleep_ms(10)

        self.write(0x06, 0)
        utime.sleep_ms(5)

        self.write(0x0C, 0)
        utime.sleep_ms(10)

    # ----------------------------------------

    def strobe(self):
        """Flashes the enable line and provides wait period."""

        self.enable_pin.value(1)
        utime.sleep_ms(1)

        self.enable_pin.value(0)
        utime.sleep_ms(1)

    # ----------------------------------------

    def write(self, command, mode):
        """Sends data to the LCD module. """

        # determine if writing a command or data
        data = command if mode == 0 else ord(command)

        # need upper nibble for first loop. lower nibble can use data directly.
        upper = data >> 4

        # write the upper nibble
```

```
        for index in range(4):
            bit = upper & 1
            self.data_bus[index].value(bit)
            upper = upper >> 1

        # strobe the LCD, sending the nibble
        self.reg_select_pin.value(mode)
        self.strobe()

        # write the lower nibble
        for index in range(4):
            bit = data & 1
            self.data_bus[index].value(bit)
            data = data >> 1

        # Strobe the LCD, sending the nibble
        self.reg_select_pin.value(mode)
        self.strobe()
        utime.sleep_ms(1)
        self.reg_select_pin.value(1)

    # ----------------------------------------

    def clear(self):
        """Clear the LCD Screen."""

        self.write(0x01, 0)
        utime.sleep_ms(5)

    # ----------------------------------------

    def home(self):
        """Return the Cursor to the starting position."""

        self.write(0x02, 0)
        utime.sleep_ms(5)

    # ----------------------------------------


    def blink(self):
        """Have the cursor start blinking."""

        self.write(0x0D, 0)
        utime.sleep_ms(1)
```

```
    # ----------------------------------------

    def cursor_on(self):
        """Have the cursor on, Good for debugging."""

        self.write(0x0E, 0)
        utime.sleep_ms(1)

    # ----------------------------------------

    def cursor_off(self):
        """Turn the cursor off."""

        self.write(0x0C, 0)
        utime.sleep_ms(1)

    # ----------------------------------------

    def print(self, string):
        """Write a string on to the LCD."""

        for element in string:
            self._putch(element)

    # ----------------------------------------

    def _putch(self, c):
        """Write a character on to the LCD."""
        self.write(c, 1)

    # ----------------------------------------

    def _puts(self, string):
        """Write a string on to the LCD."""

        for element in string:
            self._putch(element)


    # ----------------------------------------
    def go_to(self, column, row):


        if row == 0:
```

```
            address = 0

        if row == 1:
            address = 0x40

        if row == 2:
            address = 0x14

        if row == 3:
            address = 0x54

        address = address + column
        self.write(0x80 | address, 0)


# ----------------------------------------
#              END OF FILE
# ----------------------------------------
```