Electronics Club

IIT KANPUR

Analog Neural Network

Summer Project 2025

Mentors

Soham Panchal
Pranika Mittal
vihaan Sapra
Vidhi Vivek Chordia

June , 2025

Contents

1 Timeline

- Introduction to Tinkercad and Arduino board and pins.
- Introduction to PCB Design and Circuit Schematic using KiCad.
- Introduction to Machine Learning : Linear Regression,Logistic Regression,K-Nearest Neighbors (KNN),Weights and Biases.
- Introduction to Neural Networks and Backpropagation.
- Introduction to Analog Electronics: Amplifiers, Operational Amplifiers.
- Introduction to LTspice
- Project implementation

2 Objective

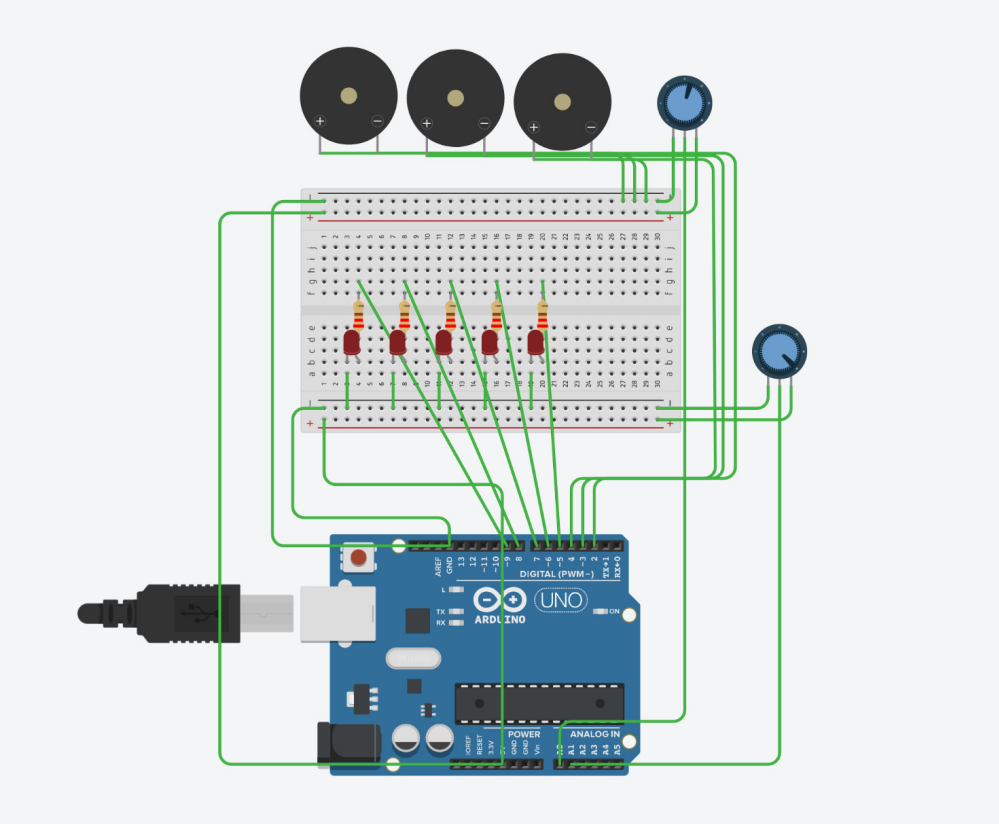
- this project aims to create an analog neural network using op-amps and resistors.

3 TinkerCAD Simulation

- Pot1 controls 3 buzzers:
  - 0–341: Buzzer 1
  - 342–682: Buzzer 2
  - 683–1023: Buzzer 3
- Pot2 controls LED delay

Connections:

- Pot1 $\rightarrow$ A0, Pot2 $\rightarrow$ A1
- Buzzers: D2, D3, D4
- LEDs: D5 to D9 (through 220Ω resistors)



3.1 Microcontroller Code

```
int leds[] = {5, 6, 7, 8, 9};
int numLeds = 5;
float t;

void setup() {
  pinMode(4, OUTPUT);
  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  Serial.begin(9600);
  for (int i = 0; i < numLeds; i++) {
    pinMode(leds[i], OUTPUT);
  }
}

void loop() {
  int POT1 = analogRead(A0);
  float POT2 = analogRead(A1);
  t = (POT2 + 113.666667)/1.1366667;
  Serial.println(t);

  if (POT1 <= 341) {
    digitalWrite(4, HIGH); digitalWrite(2, LOW); digitalWrite(3, LOW);
  }
  else if (POT1 <= 682) {
```

```
    digitalWrite(4, LOW); digitalWrite(2, HIGH); digitalWrite(3, LOW);
  }
  else {
    digitalWrite(4, LOW); digitalWrite(2, LOW); digitalWrite(3, HIGH);
  }

  for (int i = 0; i < numLeds; i++) {
    digitalWrite(leds[i], HIGH);
    delay(t);
    digitalWrite(leds[i], LOW);
  }
}
```

4 PCB Design and Schematic Circuit Design Using KiCad

4.1 Introduction

As part of the summer project on Analog Neural Networks, we were introduced to schematic design and printed circuit board (PCB) layout using the open-source software **KiCad**. This tool allows us to digitally construct electronic circuits, simulate their behavior, and ultimately design a manufacturable PCB layout.

4.2 Schematic Circuit Design

The schematic editor in KiCad provides a graphical interface to represent the electronic circuit using symbols for components such as resistors, capacitors, op-amps, transistors, and connectors. Key steps include:

1. **Placing Components:** Components are selected from the KiCad library and placed on the canvas.
2. **Wiring Connections:** Electrical connections are made using the wire tool to connect component pins logically.
3. **Annotating and Assigning Footprints:** Each component is assigned a unique label and a corresponding physical footprint.
4. **Electrical Rule Check (ERC):** The circuit is verified for common mistakes like unconnected pins or conflicting signals.

4.3 PCB Layout Design

Once the schematic is complete and error-free, the netlist is transferred to the PCB layout editor. The steps involved are:

1. **Component Placement:** Components are arranged on the board in a way that minimizes wiring complexity and respects physical constraints.
2. **Routing Traces:** Electrical connections from the schematic are routed using copper tracks.
3. **Defining Board Outline:** The physical dimensions of the PCB are defined.
4. **DRC Check:** A Design Rule Check ensures that no rules (e.g., minimum clearance or track width) are violated.
5. **Generating Gerber Files:** The final step involves generating Gerber files, which are used for PCB manufacturing.

4.4 Relevance to Analog Neural Networks

PCB and schematic design skills are essential in developing analog neural network hardware. Designing amplifier stages, summing circuits, and bias networks on a PCB allows for:

- Compact and reliable implementation of analog neural blocks
- Easy prototyping and debugging using test points and connectors
- Integration with microcontroller-based control systems or signal acquisition tools

4.5 Hands-on Schematic Implementation in KiCad

In this section, we describe the practical steps taken in KiCad to design the schematic shown below. The circuit integrates an **Arduino Nano**, three **push buttons**, an **LED indicator**, and two I^2C peripherals: the **MAX30102 pulse sensor** and **SSD1306 OLED display**.

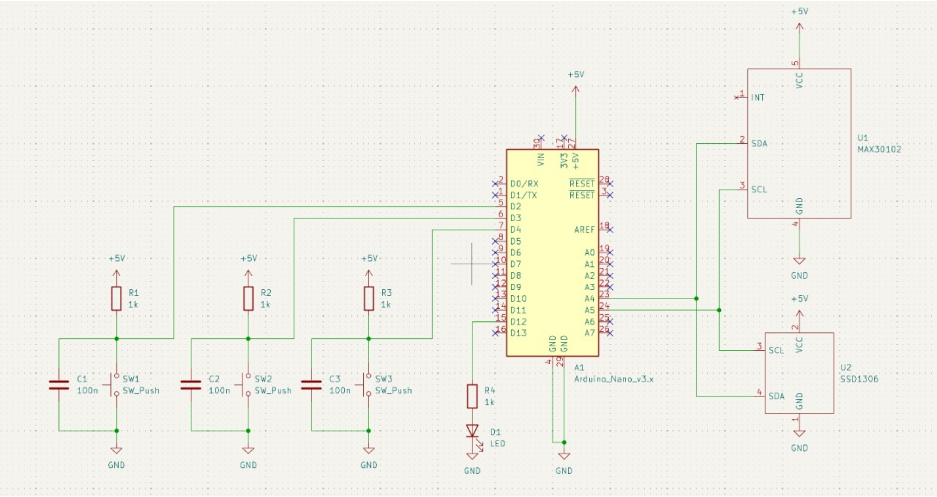


Figure 1: Schematic design in KiCad

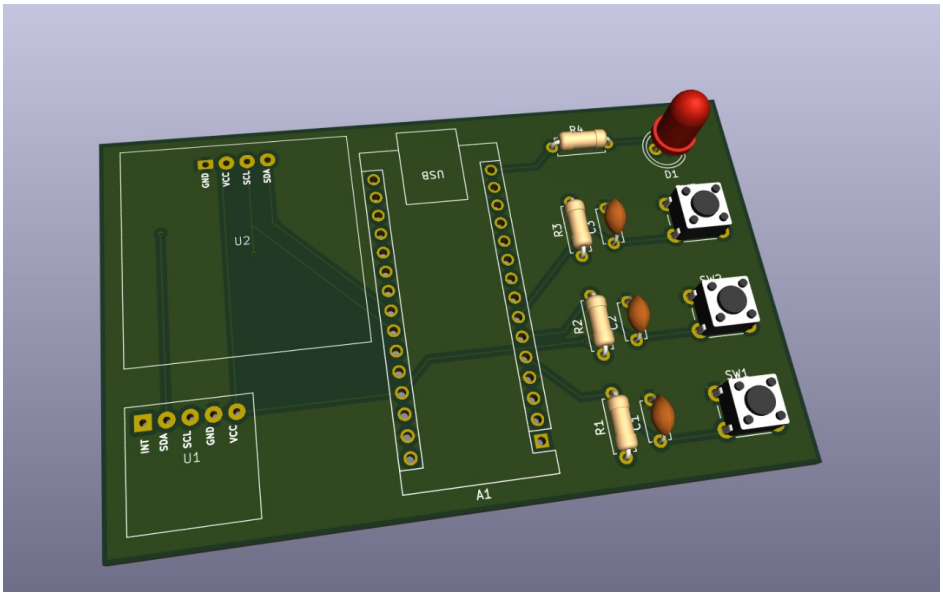


Figure 2: PCB layout in KiCad

4.5.1 Placing and Annotating Components

- **Arduino Nano** microcontroller was added from the library.
- **Push buttons** SW1, SW2, SW3 were placed and each connected with 1kΩ resistor (R1-R3) and 100nF capacitor (C1-C3) for debounce filtering.
- An **LED** (D1) and a resistor (R4) were added for visual feedback.
- I^2C devices (**MAX30102** and **SSD1306**) were added using 4-pin header footprints.

4.5.2 Wiring Connections

- SW1 → D2, SW2 → D3, SW3 → D4 on Arduino
- LED connected to pin D11 through resistor R4
- I^2C: SDA → A4, SCL → A5
- All VCC and GND lines were routed using power symbols

4.5.3 Assigning Footprints

Each component was linked to a physical footprint for PCB layout:

- Nano: 2×15 header DIP
- Buttons: THT tactile switch
- Capacitors, Resistors: THT axial

- LED: 5mm THT
- Sensors: 4-pin header

4.5.4 ERC Check

ERC (Electrical Rules Check) ensured no floating pins or missing connections. KiCad flagged minor warnings which were addressed or acknowledged.

4.5.5 PCB Layout (Brief Overview)

- Nano centered for routing convenience
- Sensors placed close for clean I^2C lines
- Traces routed manually with DRC compliance
- Board outline defined to fit a standard module form

4.6 Conclusion

Learning KiCad equips us with practical hardware design skills, bridging the gap between theoretical circuit analysis and real-world implementation. In future phases of the project, we aim to fabricate and test a custom PCB containing analog neuron components designed during the schematic phase.

5 Linear Regression

5.1 Definition

Linear regression is a type of machine-learning algorithm that learns from the labelled datasets and maps the data points with most optimized linear functions which can be used for prediction on new datasets. It assumes that there is a linear relationship between the input and output

5.2 Mathematics of Linear Regression

The basic form of a linear regression model is:

$$y = \beta_0 + \beta_1 x + \epsilon$$

Where:

- y is the dependent variable (target)
- x is the independent variable (feature)
- β_0 is the intercept

- β_1 is the coefficient (slope)
- ϵ is the error term

In matrix form for multiple features:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

The goal is to minimize the cost function:

$$J(\boldsymbol{\beta}) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\beta(x^{(i)}) - y^{(i)} \right)^2$$

Where $h_\beta(x) = x^T \beta$ is the hypothesis.

5.3 Implementation in Python

The implementation of linear regression includes data loading, preprocessing, visualization, model training (custom and scikit-learn), and evaluation.

Step 1: Data Loading and Preprocessing

```
from google.colab import files
files.upload()  # Upload 'Real estate.csv'

import pandas as pd
df = pd.read_csv("Real estate.csv")
```

Checking for missing values:

```
print(df.isnull().sum())
```

Step 2: Data Visualization

Visualizing the relationship between each feature and the target:

```
import matplotlib.pyplot as plt

for col in df.columns[1:-1]:
    plt.figure(figsize=(6, 4))
    plt.scatter(df[col], df['Y house price of unit area'])
    plt.title(f'{col} vs Y house price of unit area')
    plt.xlabel(col)
    plt.ylabel('Y house price of unit area')
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

Step 3: Feature Selection and Splitting

Removing non-predictive index column and splitting the data:

```
df = df.drop(columns=['No'])

from sklearn.model_selection import train_test_split
train_data, test_data = train_test_split(df, test_size=0.25, random_state
    =52)
```

Step 4: Feature Scaling

Scaling features using Min-Max normalization:

```
from sklearn.preprocessing import MinMaxScaler

X_train = train_data.drop(columns=['Y house price of unit area'])
y_train = train_data['Y house price of unit area']

X_test = test_data.drop(columns=['Y house price of unit area'])
y_test = test_data['Y house price of unit area']

scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Step 5: Custom Linear Regression with Gradient Descent

Training a custom linear regression model:

```
model = LinearRegression(learning_rate=0.87, epochs=4023)
model.fit(X_train_scaled, y_train.values)
y_pred = model.predict(X_test_scaled)
```

Step 6: Model Evaluation

Evaluating with MSE and R^2 score:

```
from sklearn.metrics import mean_squared_error, r2_score

mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"MSE: {mse:.4f}")
print(f"R$^2$: {r2:.4f}")
```

Step 7: Comparison with Scikit-learn's LinearRegression

Comparing with the standard scikit-learn model:

```
from sklearn.linear_model import LinearRegression as SKLinearRegression

sk_model = SKLinearRegression()
sk_model.fit(X_train_scaled, y_train)
y_pred_sk = sk_model.predict(X_test_scaled)

mse_sk = mean_squared_error(y_test, y_pred_sk)
r2_sk = r2_score(y_test, y_pred_sk)

print(f"Sklearn LinearRegression - MSE: {mse_sk:.4f}")
print(f"Sklearn LinearRegression - R$^2$: {r2_sk:.4f}")
```

6 Logistic Regression

6.1 Definition

Logistic regression is a supervised machine learning algorithm used for binary classification tasks. Instead of predicting a continuous output, it predicts the probability that a given input belongs to a particular class using a sigmoid (logistic) function.

6.2 Mathematics of Logistic Regression

Unlike linear regression, logistic regression uses the sigmoid function to squash output values between 0 and 1:

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \text{where } z = \beta_0 + \beta_1 x$$

The hypothesis becomes:

$$h_\beta(x) = \frac{1}{1 + e^{-x^T \beta}}$$

The cost function is derived from likelihood estimation (cross-entropy loss):

$$J(\boldsymbol{\beta}) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(h_\beta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\beta(x^{(i)})) \right]$$

We minimize this using gradient descent or other optimization techniques.

6.3 Implementation of Logistic Regression

We implemented binary classification using logistic regression on the Breast Cancer dataset available in `scikit-learn`. The steps include loading the dataset, preprocessing, implementing logistic regression from scratch, and evaluating performance.

Step 1: Load and Preprocess Data

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import pandas as pd
import numpy as np

data = load_breast_cancer()
X = data.data
y = data.target

scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.25, random_state=52
)
```

Step 2: Define Sigmoid Function and its Derivative

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(z):
    return sigmoid(z) * (1 - sigmoid(z))
```

Step 3: Define Custom Logistic Regression Class

```
class LogisticRegression:
    def __init__(self, learning_rate, epochs):
        self.lr = learning_rate
        self.epochs = epochs

    def fit(self, X, y):
        n_samples, n_features = X.shape
        y = y.reshape(-1, 1)
        self.weights = np.zeros((n_features, 1))
        self.bias = np.zeros((1, 1))

        for _ in range(self.epochs):
            z = np.dot(X, self.weights) + self.bias
            y_pred = sigmoid(z)

            dw = np.dot(X.T, (y_pred - y)) / n_samples
            db = np.sum(y_pred - y) / n_samples

            self.weights -= self.lr * dw
            self.bias -= self.lr * db

    def predict(self, X):
```

```
        linear_model = np.dot(X, self.weights) + self.bias
        y_pred = sigmoid(linear_model)
        return (y_pred >= 0.5).astype(int).flatten()
```

Step 4: Train and Evaluate the Custom Model

```
from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix

model = LogisticRegression(learning_rate=0.1, epochs=5000)
model.fit(X_train, y_train)
y_pred_custom = model.predict(X_test)

acc_custom = accuracy_score(y_test, y_pred_custom)
print("Accuracy:", acc_custom)
print("Classification Report:")
print(classification_report(y_test, y_pred_custom))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_custom))
```

Step 5: Comparison with Scikit-learn Logistic Regression

```
from sklearn.linear_model import LogisticRegression as
    SKLogisticRegression

sk_model = SKLogisticRegression()
sk_model.fit(X_train, y_train)
y_pred_sk = sk_model.predict(X_test)

print("Classification Report:")
print(classification_report(y_test, y_pred_sk))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_sk))
```

7 K-Nearest Neighbors (KNN)

7.1 Definition

K-Nearest Neighbors (KNN) is a simple, instance-based, supervised learning algorithm used for both classification and regression. It predicts the output for a new data point by looking at the **K** closest training examples in the feature space and using majority vote (for classification) or average (for regression).

7.2 Mathematics of KNN

For classification, KNN does the following:

1. Compute the distance (usually Euclidean) from the new point to all training data points:

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

2. Select the K points with the smallest distance.

3. Determine the most common class among them.

KNN has no training phase — it memorizes the data and calculates distances at prediction time, making it lazy af but accurate if tuned right.

7.3 Implementation in Python

Step 1: Define distance metric

```
import numpy as np
from collections import Counter  # For majority voting

def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2)**2))
```

Step 2: Define the KNN class

```
class KNN:
    def __init__(self, k):
        self.k = k
```

Step 3: Store training data

```
    def fit(self, x_train, y_train):
        self.x_train = x_train
        self.y_train = y_train
```

Step 4: Predict labels for test data

```
    def predict(self, x_test):
        predictions = [self._helper(x) for x in x_test]
        return np.array(predictions)
```

Step 5: Find nearest neighbors and vote

```
    def _helper(self, x):
        distances = [euclidean_distance(x, x1) for x1 in self.x_train]
        indices = np.argsort(distances)[:self.k]
        labels = [self.y_train[i] for i in indices]
        majority_vote = Counter(labels).most_common(1)
        return majority_vote[0][0]
```

Step 6: Define accuracy calculation

```
def accuracy(predictions, y_test):
    return np.sum(predictions == y_test) / len(y_test)
```

Step 7: Load dataset and prepare splits

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap as lcm

colormap = lcm(['red', 'blue', 'yellow'])

iris = datasets.load_iris()
x, y = iris.data, iris.target
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
```

Step 8: Train and evaluate the model

```
clf = KNN(k=3)
clf.fit(x_train, y_train)
predictions = clf.predict(x_test)
print(accuracy(predictions, y_test))
```

Step 9: Visualize the dataset

```
plt.scatter(x[:, 0], x[:, 1], c=y, cmap=colormap)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Iris Dataset')
plt.show()
```

8 Neural Networks

8.1 Definition

A Neural Network is a computational model inspired by the structure of the human brain. It consists of layers of interconnected units (neurons) that transform input data through learned weights and non-linear activation functions. Neural networks are highly flexible and can approximate complex functions for tasks like image classification, language modeling, and more.

8.2 Mathematics of a Basic Neural Network (2 Hidden Layers)

Let the network have:

- Input: $\mathbf{x} \in \mathbb{R}^n$
- Hidden Layer 1: weights $\mathbf{W}^{[1]}, \mathbf{b}^{[1]}$, activation $\mathbf{a}^{[1]}$
- Hidden Layer 2: weights $\mathbf{W}^{[2]}, \mathbf{b}^{[2]}$, activation $\mathbf{a}^{[2]}$
- Output Layer: weights $\mathbf{W}^{[3]}, \mathbf{b}^{[3]}$, output $\hat{\mathbf{y}}$

Forward pass equations:

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}, \quad \mathbf{a}^{[1]} = \sigma(\mathbf{z}^{[1]})$$
$$\mathbf{z}^{[2]} = \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}, \quad \mathbf{a}^{[2]} = \sigma(\mathbf{z}^{[2]})$$
$$\mathbf{z}^{[3]} = \mathbf{W}^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]}, \quad \hat{\mathbf{y}} = \text{softmax}(\mathbf{z}^{[3]})$$

8.3 Backpropagation

Backpropagation is the learning algorithm used to train neural networks. It efficiently computes the gradient of the loss function with respect to each weight by applying the chain rule in reverse from the output layer back to the input.

Let L be the loss function, e.g., cross-entropy for classification. The gradient updates are:

$$\delta^{[3]} = \hat{\mathbf{y}} - \mathbf{y} \quad \text{(output layer error)}$$
$$\delta^{[2]} = \left(\mathbf{W}^{[3]T}\delta^{[3]}\right) \circ \sigma'(\mathbf{z}^{[2]})$$
$$\delta^{[1]} = \left(\mathbf{W}^{[2]T}\delta^{[2]}\right) \circ \sigma'(\mathbf{z}^{[1]})$$

Here:

- $\circ$ denotes element-wise multiplication
- $\sigma'(\cdot)$ is the derivative of the activation function

Gradients of the loss with respect to weights and biases:

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \delta^{[l]} \cdot (\mathbf{a}^{[l-1]})^T$$
$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \delta^{[l]}$$

Weights are then updated using gradient descent:

$$\mathbf{W}^{[l]} := \mathbf{W}^{[l]} - \eta \frac{\partial L}{\partial \mathbf{W}^{[l]}}$$

Where η is the learning rate.

8.4 Implementation

The handwritten digit classification task was implemented in Python using the `TensorFlow Keras` library with the MNIST dataset. This dataset consists of 60,000 training images and 10,000 testing images, each of size 28×28 pixels in grayscale.

8.4.1 Data Preparation

The dataset was loaded and visualized to ensure correctness. Each image was flattened into a one-dimensional vector of 784 pixels. The training data was split into:

- Training set
- Validation set (10,000 samples)
- Test set

8.4.2 Model Architecture

A fully connected feedforward neural network was designed with the following layers:

- Input layer: 784 neurons (flattened pixel values)
- Hidden layer 1: 128 neurons, ReLU activation
- Hidden layer 2: 64 neurons, ReLU activation
- Output layer: 10 neurons, Softmax activation

8.4.3 Training Configuration

The model was compiled using the Adam optimizer with sparse categorical cross-entropy loss. Training was performed for 10 epochs with a batch size of 512, and validation accuracy was monitored throughout.

8.4.4 Python Code

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

# Load dataset
mnist = keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Visualize samples
n = 5
index = np.random.choice(x_train.shape[0], n)
for i, ind in enumerate(index):
    plt.subplot(1, n, i+1)
    plt.imshow(x_train[ind], cmap="gray")
    plt.axis("off")
plt.show()

# Flatten images
x_train_flatten = x_train.reshape(x_train.shape[0], -1)
x_test_flatten = x_test.reshape(x_test.shape[0], -1)
```

```
# Create validation set
n_validation = 10000
x_validation = x_train_flatten[:n_validation]
y_validation = y_train[:n_validation]
x_train_flatten = x_train_flatten[n_validation:]
y_train = y_train[n_validation:]

# Define model
model = keras.models.Sequential([
    layers.Input(shape=(784,)),
    layers.Dense(128, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Compile model
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

# Train model
history = model.fit(x_train_flatten, y_train,
                    epochs=10, batch_size=512,
                    validation_data=(x_validation, y_validation))

# Evaluate on test set
results = model.evaluate(x_test_flatten, y_test)
print("Loss =", results[0])
print("Accuracy =", results[1] * 100, "%")

# Plot accuracy
plt.plot(history.history['accuracy'], label="Training accuracy")
plt.plot(history.history['val_accuracy'], label="Validation accuracy")
plt.title("Model accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()

# Plot loss
plt.plot(history.history['loss'], label="Training loss")
plt.plot(history.history['val_loss'], label="Validation loss")
plt.title("Model loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()

# Predictions
n = 5
index = np.random.choice(x_test.shape[0], n, replace=False)
for i, ind in enumerate(index):
    plt.subplot(1, n, i+1)
```

```
    plt.imshow(x_test[ind].squeeze(), cmap='gray')
    plt.axis("off")
plt.show()

for ind in index:
    image = x_test_flatten[ind].reshape(1, -1)
    prediction = model.predict(image, verbose=0)
    print(np.argmax(prediction), end=" ")
```

8.4.5 Results

The model achieved high classification accuracy on the MNIST test set. Accuracy and loss plots confirmed stable convergence with minimal overfitting.

9 Amplifiers and Operational Amplifiers

9.1 Voltage Amplifiers

Amplifiers are essential electronic devices that increase the amplitude of an input signal. A basic voltage amplifier is characterized by three key parameters:

- **Input Resistance (R_i):** Ideally infinite, to prevent loading of the source.
- **Output Resistance (r_o):** Ideally zero, for maximum power transfer.
- **Transconductance (g_m):** The gain factor, defined as I_o/V_i.

For a voltage amplifier using a transistor-like device:

$$V_o = -g_m r_o V_i$$

The voltage gain becomes:

$$A_v = \frac{V_o}{V_s} = -g_m r_o \cdot \frac{R_L}{R_S + R_i}$$

To achieve effective amplification, the condition $g_m r_o \gg 1$ must be satisfied.

9.2 Biasing and Signal Integrity

Since real devices do not behave like ideal transistors over their entire operating range, biasing is necessary to push them into a region suitable for amplification. A carefully chosen DC bias point ensures that:

- The signal is not clipped or distorted.
- Unwanted DC components at the output are removed using capacitive coupling.

9.3 Operational Amplifiers (Op-Amps)

Operational amplifiers are semi-custom analog building blocks designed for ease of use. They exhibit:

- Very high differential gain
- Very high input resistance
- Very low output resistance
- High Common Mode Rejection Ratio (CMRR)

An ideal op-amp model assumes:

$$\begin{aligned} A_{OL} &\to \infty \\ R_i &\to \infty \\ R_o &\to 0 \\ v_o &= A_{OL} \cdot (v_+ - v_-) \end{aligned}$$

9.4 Op-Amp Configurations

Two commonly used op-amp configurations in neural circuits include:

- **Inverting Amplifier:**
$$V_o = -\frac{R_2}{R_1} V_s$$
- **Non-Inverting Amplifier:**
$$V_o = \left(1 + \frac{R_2}{R_1}\right) V_s$$

These make use of the virtual ground concept, which assumes $v_+ = v_-$ and $i_{\text{in}} = 0$ under negative feedback conditions.

9.5 Adder Circuits

An **adder circuit** is an operational amplifier configuration that produces an output voltage proportional to the algebraic sum of two or more input voltages. It is often implemented in the inverting configuration but can be realized in more general forms to incorporate weighting factors.

9.5.1 Principle of Operation

In this configuration:

- Each input voltage is applied through a resistor network to either the inverting or non-inverting terminal of the op-amp.
- The non-inverting terminal may also be connected through a resistor network to ground to control biasing.
- Due to the **virtual ground property** in an ideal op-amp with negative feedback, the inverting input voltage is held at approximately zero volts.
- The combined effect of currents from the different inputs produces the summed output voltage.

9.5.2 General Equation

The detailed node-voltage analysis yields:

$$v_o = v_{s1} \cdot \frac{\frac{R_2 R_3}{R_2+R_3}}{\frac{R_2 R_3}{R_2+R_3} + R_1} \cdot \left(1 + \frac{R_f}{R_4}\right) + v_{s2} \cdot \frac{\frac{R_1 R_3}{R_1+R_3}}{\frac{R_1 R_3}{R_1+R_3} + R_2} \cdot \left(1 + \frac{R_f}{R_4}\right)$$

This is sometimes referred to as the **high-entropy expression** due to its complexity.

For compactness, we can define:

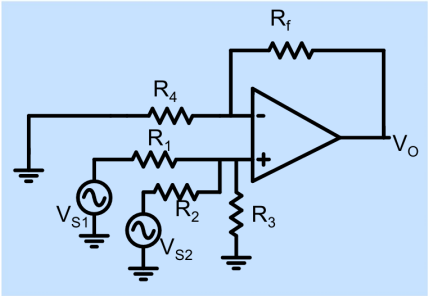
$$R_P = R_1 \parallel R_2 \parallel R_3$$

and rewrite as the **low-entropy expression**:

$$v_o = \left(\frac{R_P}{R_1} v_{s1} + \frac{R_P}{R_2} v_{s2}\right) \cdot \left(1 + \frac{R_f}{R_4}\right)$$

9.5.3 Circuit Diagram

Adder



$$v_o = v_{s1}\frac{R_2 \| R_3}{R_2 \| R_3 + R_1} \times (1 + \frac{R_f}{R_4})$$

$$+ v_{s2}\frac{R_1 \| R_3}{R_1 \| R_3 + R_2} \times (1 + \frac{R_f}{R_4})$$

High entropy expression !

$$R_P = R_1 \| R_2 \| R_3$$

$$v_o = (\frac{R_p}{R_1} v_{s1} + \frac{R_p}{R_2} v_{s2}) \times (1 + \frac{R_f}{R_4})$$

Low entropy expression !

9.6 Subtractor Circuits

A **subtractor circuit** outputs a voltage proportional to the difference between two input voltages. This configuration is widely used for differential measurements and common-mode rejection.

9.6.1 Principle of Operation

- One input is applied to the inverting terminal through a resistor R_1.
- The second input is applied to the non-inverting terminal via a voltage divider (R_2 and R_3).
- The op-amp amplifies the difference between these terminals, and resistor ratios control scaling.
- Proper resistor matching ensures high **common-mode rejection**.

9.6.2 General Equation

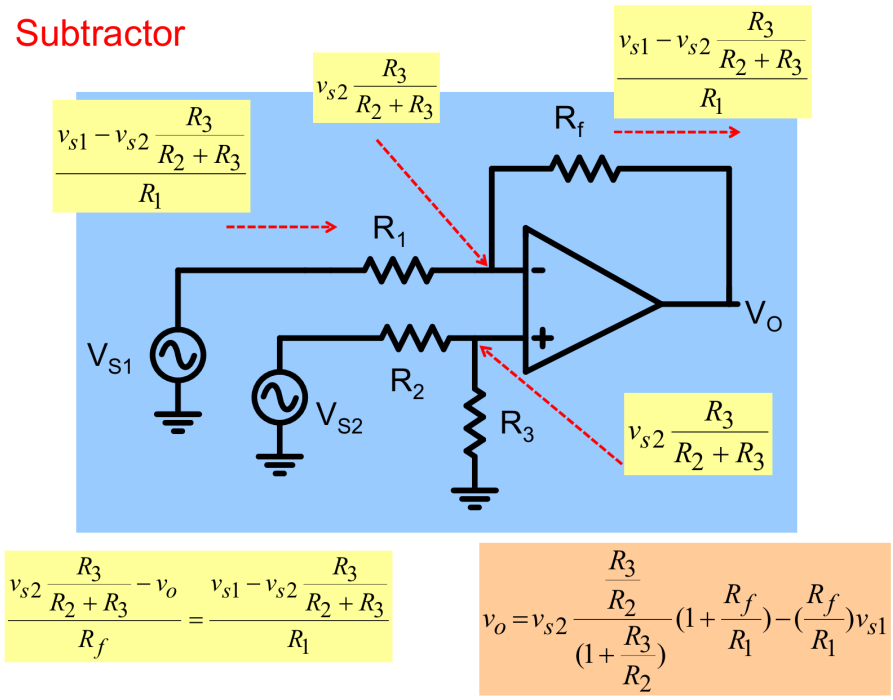
By analyzing the circuit:

$$v_o = v_{s2} \cdot \frac{\frac{R_3}{R_2+R_3}}{\left(1 + \frac{R_3}{R_2}\right)} \cdot \left(1 + \frac{R_f}{R_1}\right) - \frac{R_f}{R_1} v_{s1}$$

This can be rewritten more compactly as:

$$v_o = v_{s2} \cdot \frac{R_3}{R_2 + R_3} \cdot \left(1 + \frac{R_f}{R_1}\right) - \frac{R_f}{R_1} v_{s1}$$

9.6.3 Circuit Diagram



9.7 Relevance to Neural Networks

In **analog neural networks**, the basic computational unit (or node) must be able to perform operations such as:

- Scaling of input signals by a given weight
- Summation of multiple weighted inputs
- Subtraction for differential processing

Operational amplifiers, configured as inverting/non-inverting amplifiers, adders, and subtractors, provide these capabilities in hardware. By carefully selecting resistor values, each op-amp stage can be programmed to implement a specific *weight* in the neural network model.

Integration into Neural Nodes

In an analog neural network:

- Each **input signal** to a neuron is first scaled by a gain factor implemented via an op-amp amplifier stage.
- The scaled signals are then combined using an **adder circuit** to perform the weighted summation required for neuron activation.
- If necessary, a **subtractor circuit** can be used to implement inhibitory connections or to compute differences between signals.
- The resulting summed signal can then be passed to an activation function block (e.g., implemented using a nonlinear circuit).

Analog Neural Network Architecture

In this approach:

1. Each node corresponds to a set of op-amp stages that realize the mathematical neuron equation:
$$y = f\left(\sum_i w_i x_i\right)$$
where w_i is the weight (set by resistor ratios) and x_i is the input signal.
2. The **adder** circuit sums all excitatory inputs, while subtractor stages handle inhibitory inputs.
3. The network operates entirely in the analog domain, meaning that signal processing occurs continuously and without digitization.

This hardware realization enables parallel, low-latency computation and can be advantageous in applications where speed and power efficiency are critical.

10 LTspice Circuit Simulation

The circuit designs for the op-amp adder and subtractor were first created and verified in **LTspice**, a free SPICE-based analog circuit simulator developed by Analog Devices. LTspice is widely used for testing and analyzing electronic circuits before physical implementation, allowing designers to visualize signal behavior, measure performance parameters, and optimize component values without the cost and time of building prototypes.

Using LTspice for Circuit Design

LTspice provides a schematic editor for drawing circuits, a library of electronic components, and a powerful simulation engine. The typical workflow for using LTspice in this project involved:

1. Launching LTspice and creating a new schematic file.

2. Placing required components (op-amps, resistors, voltage sources) from the built-in library.

3. Wiring the components together to form the adder and subtractor configurations.

4. Assigning realistic resistor values to represent the neural network weights.

5. Setting up the simulation type (e.g., transient analysis for time-domain signals, AC analysis for frequency response).

6. Running the simulation and observing the waveforms at the output nodes.

Through simulation, the theoretical functionality of both circuits was confirmed. This ensured that the chosen resistor ratios and op-amp configurations behaved as expected before proceeding to PCB design and hardware assembly.

11 Project Implementation

11.1 Introduction

This project focuses on the design and training of a **Neural Network using analog circuits**. In the implemented design, the weights of the network are stored as voltages in capacitors within op-amp integrator circuits.

For each training data sample, the circuit directly adjusts the weight combination such that the actual output approaches the expected output. After multiple training iterations, all weight values are combined and stored as a vector, with their average treated as the optimal solution.

11.2 Neurons

Each neuron in the analog network performs two key functions:

1. Summation of input signals.

2. Application of a nonlinear activation function.

The summation is implemented using an **op-amp adder** circuit (see Fig. 3).

The chosen activation function is the **Rectified Linear Unit (ReLU)**, shown in Fig. 4. In our implementation:

- The lower limit of the activation function includes a small positive offset.

- The upper limit is constrained by the op-amp supply voltage.

A **Buffer ReLU** stage is added after the activation to prevent loading effects on the previous stage.

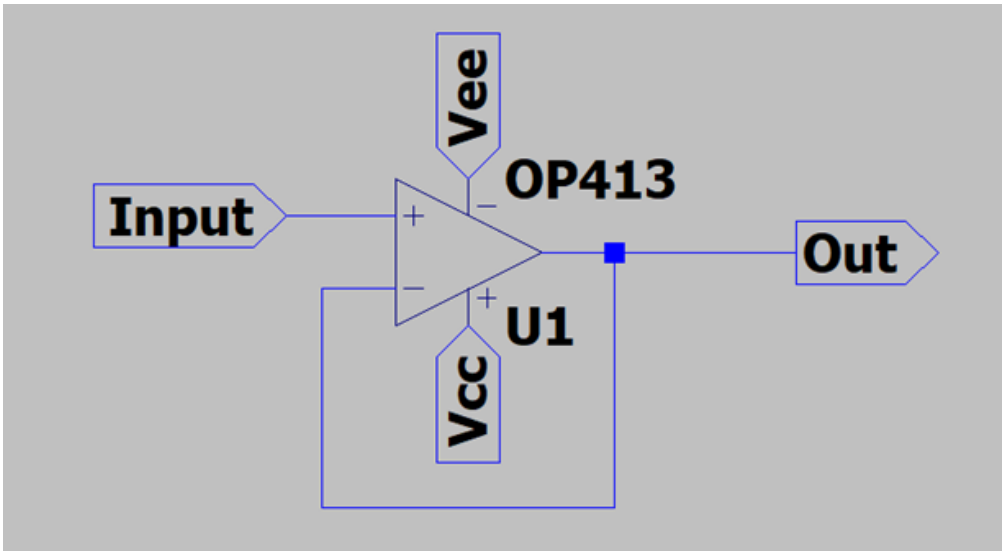


Figure 3: Op-amp adder circuit used for neuron summation.

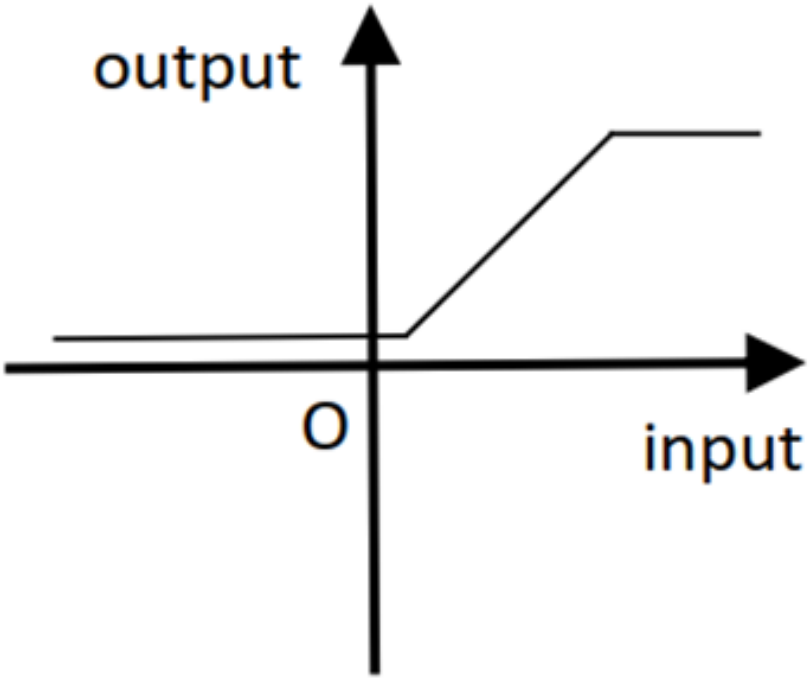


Figure 4: ReLU activation function with offset and saturation limits.

11.3 Synapses

Synapses store the connection weights and multiply them with the outputs of the previous neurons. In this design:

- A capacitor in the op-amp integrator stores the voltage corresponding to a weight.
- The integrator output voltage V_w represents the stored weight.
- The charging rate (training speed) is controlled by an adjustment voltage V_{adjust}.
- A zero adjustment voltage holds the weight constant.

The relation between V_w and V_{adjust} can be expressed as:

$$V_w = \frac{1}{R_3 C_1} \int_0^t \frac{R_2}{R_1} V_{adjust}\, dt + C \tag{1}$$

The multiplication of the stored weight with the input signal is performed by an **analog multiplier IC** (AD633 in this case).

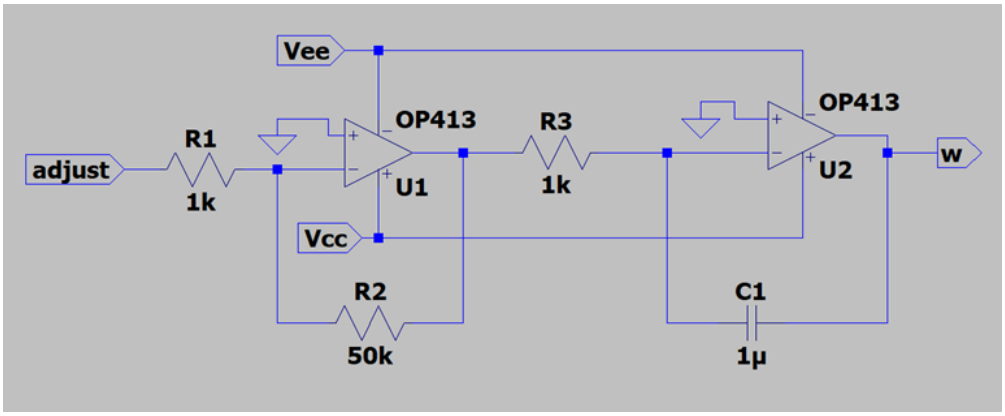


Figure 5: Synapse circuit using integrator and analog multiplier.

11.4 Feedback Circuit

While neurons and synapses implement the feedforward path of the ANN, training requires a feedback mechanism to adjust the weights.

The feedback circuit compares:

- Expected output voltage (V_{ex})
- Actual output voltage (V_{ac})

The difference between these voltages generates V_{adjust}, which updates the weights. A **voltage follower** is used in the feedback path to buffer the signals.

To prevent oscillations caused by the op-amp's high gain and ANN delays:

- Additional feedback is used to reduce error gain.
- Resistors R_1 and R_2 in the feedback path control the gain.

The gain control equation is:

$$V_{adjust} = \frac{R_1}{R_2}(V_{ex} - V_{ac}) + V_{ac} \tag{2}$$

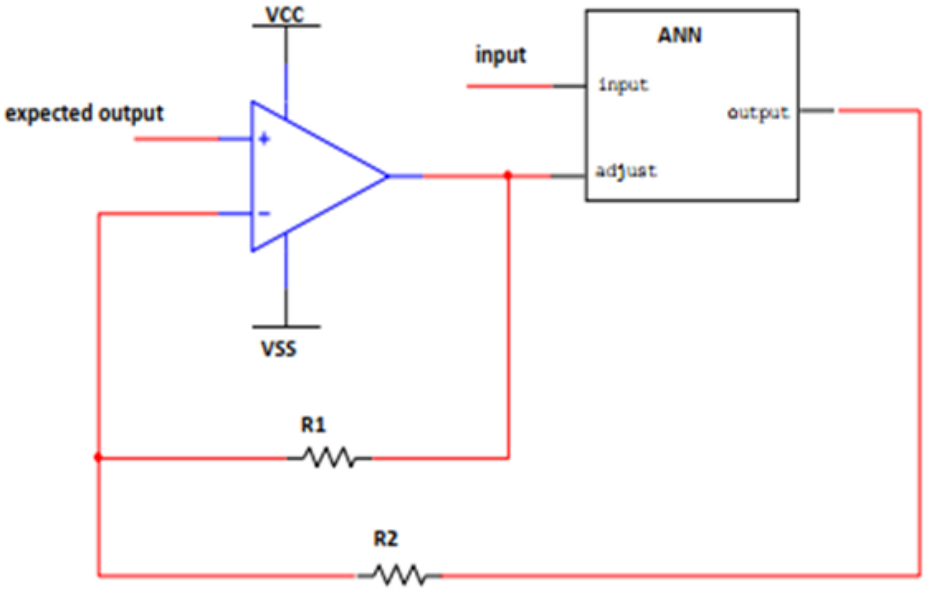


Figure 6: Feedback circuit with gain control for weight adjustment.

11.5 Full Circuit and Neuron Trainer

The complete analog neural network node integrates:

- Neuron summation and activation
- Synapse weight storage and multiplication
- Feedback-based weight adjustment

The "WEIGHT" block corresponds to the integrator and multiplier combination that stores and applies connection weights.

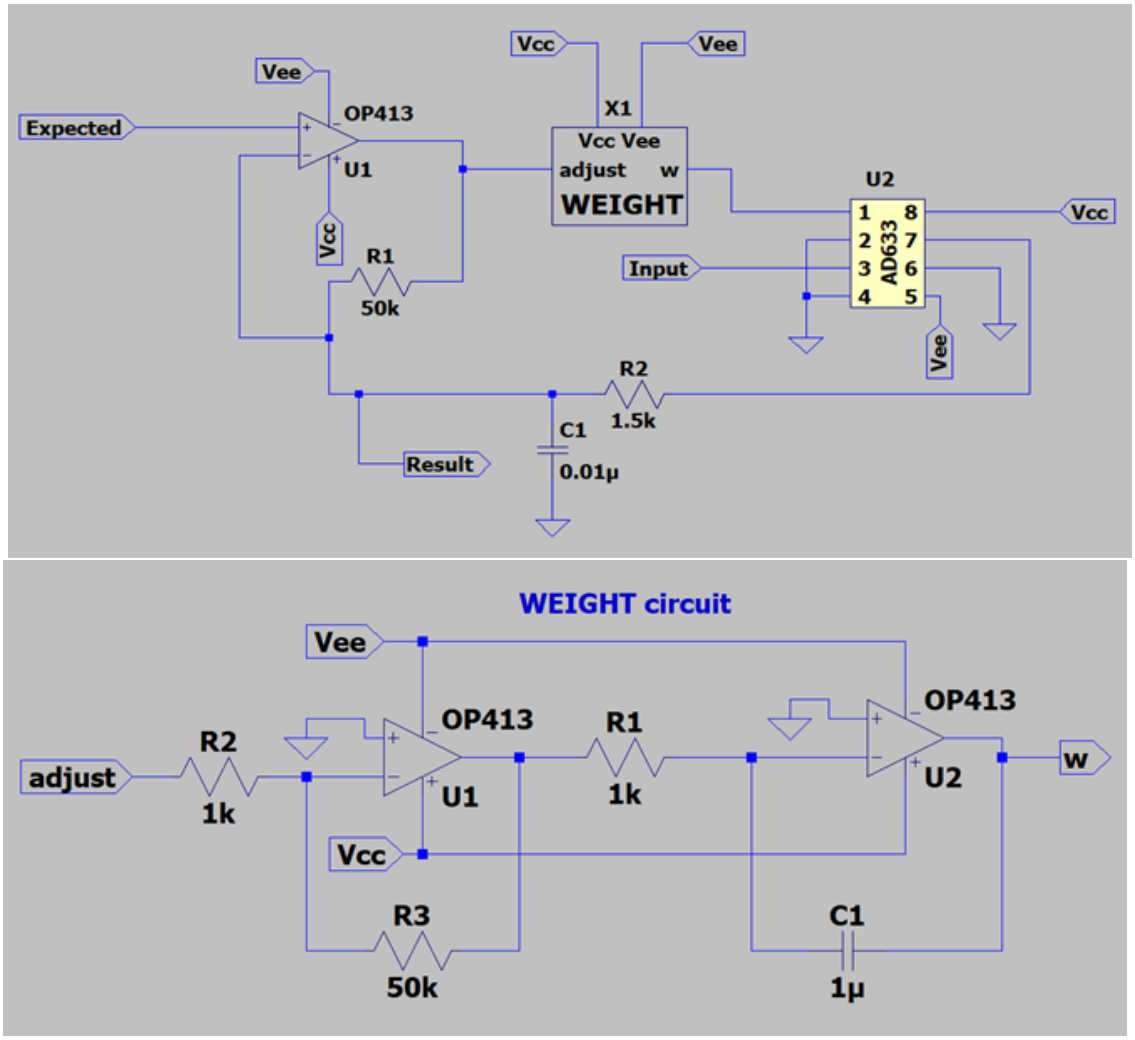


Figure 7: Full analog neuron trainer circuit.

11.6 Simulation Results

All circuit simulations were performed in **LTspice**.

For an input signal of 5 V:

- The red curve represents the calculated weight for the neuron.
- The green curve is the expected output.
- The blue curve is the actual output.

The simulation shows that the neuron rapidly trains, adjusting the stored weight so that the actual output closely follows the expected output.

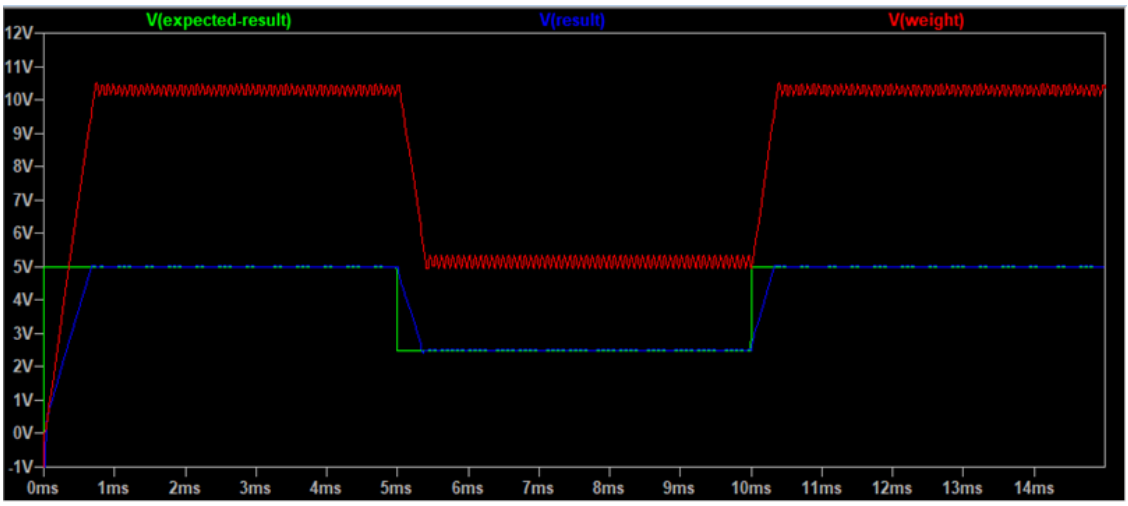


Figure 8: LTspice simulation showing neuron weight adjustment and output tracking.

References

[1] A. S. Sedra and K. C. Smith, *Microelectronic Circuits*, 7th ed., Oxford University Press, 2015.

[2] Texas Instruments, "Op Amps for Everyone," Application Report, 2017. [Online]. Available: `https://www.ti.com/lit/an/slod006b/slod006b.pdf`

[3] S. Yu, "Neuro-inspired computing with emerging nonvolatile memory," *Proceedings of the IEEE*, vol. 106, no. 2, pp. 260–285, Feb. 2018.

[4] P. Horowitz and W. Hill, *The Art of Electronics*, 3rd ed., Cambridge University Press, 2015.

[5] KiCad EDA, "KiCad Electronic Design Automation," 2023. [Online]. Available: `https://www.kicad.org`

[6] Autodesk, "Tinkercad Circuits," 2023. [Online]. Available: `https://www.tinkercad.com/circuits`

[7] B. Linares-Barranco and T. Serrano-Gotarredona, "Memristance can explain Spike-Time-Dependent-Plasticity in neural synapses," *Nature Precedings*, 2009.

[8] Y. Zhang, X. Wu, and X. Liu, "Analog Neural Network Implementation Using CMOS and Memristor Circuits," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 9, pp. 3028–3039, Sep. 2020. [Online]. Available: `https://ieeexplore.ieee.org/document/9018997`

[9] A. Basu, "Hardware Neural Networks: Concepts, Architectures and Applications," in *Emerging Research in Electronics, Computer Science and Technology*, Springer,

2014, pp. 75–84. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-23866-6_8

[10] B. Razavi, *Fundamentals of Microelectronics*, 2nd ed., Wiley, 2014.