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.

2 Objective

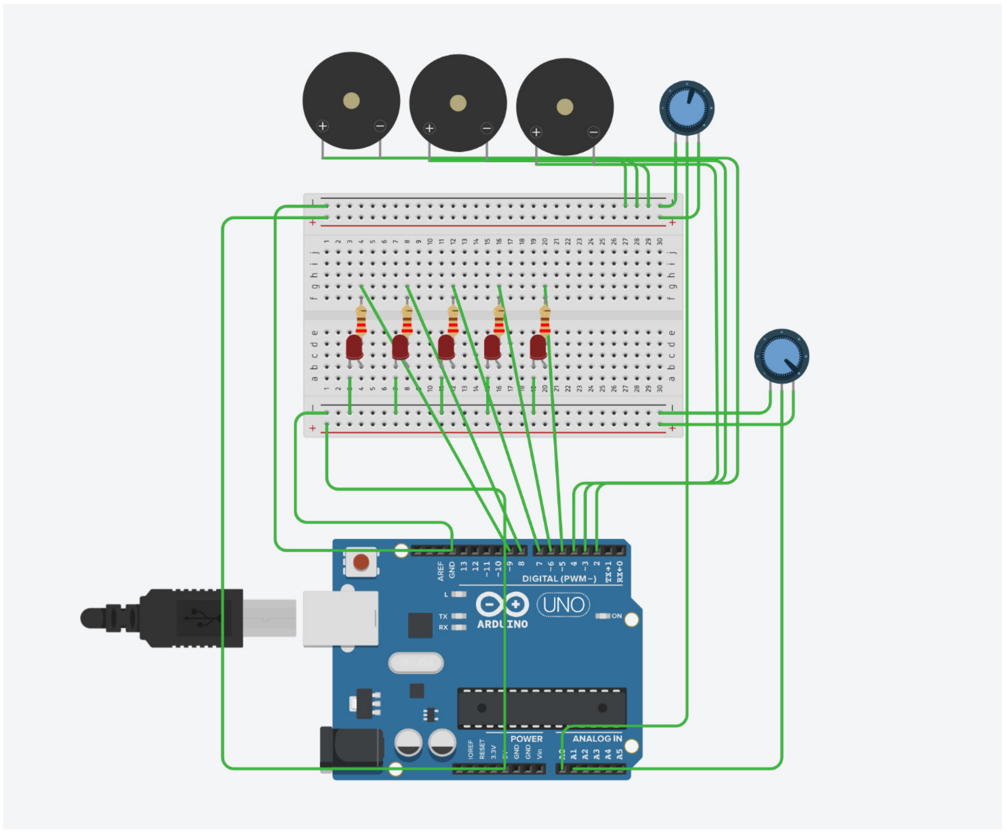
- this project aims to create an analog neural network using op-amps and capacitors, and resistors.

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)



3.0.1 Arduino 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^{2}C peripherals: the **MAX30102 pulse sensor** and **SSD1306 OLED display**.

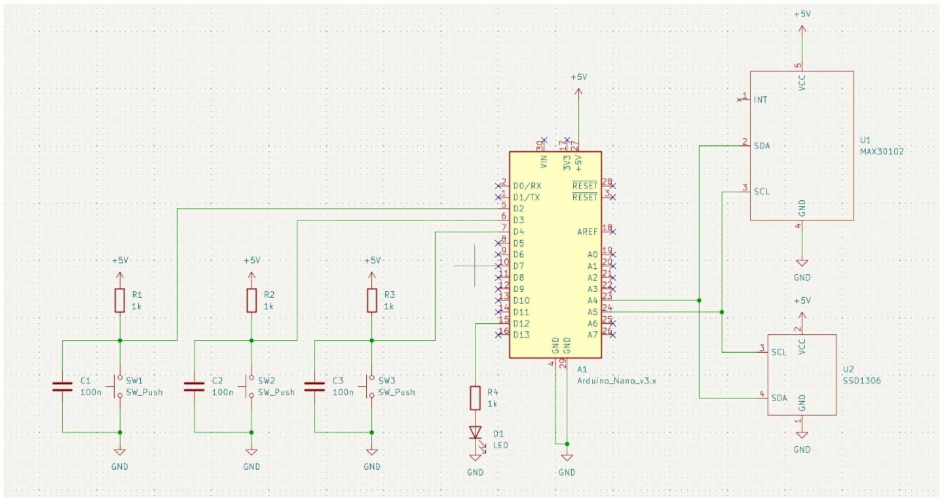


Figure 1: Schematic design in KiCad

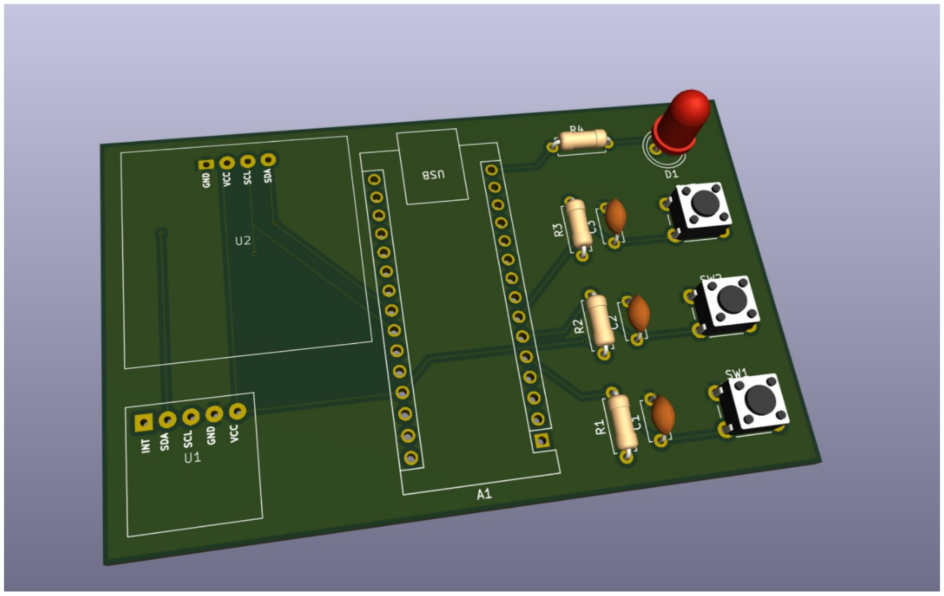


Figure 2: PCB layout in KiCad

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^{2}C 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  : {r2:.4f}")
```

Step 7: Comparison with Scikit-learn's LinearRegression

Comparing with the standard scikit-learn model:

```python
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  : {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

```python
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

```python
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

```python
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: $\mathbf{x} \in R^n$
- 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:

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

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:

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

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:

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

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.

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 and Subtractor Circuits

Op-amps can also be configured to perform linear operations such as:

- **Adder:** $V_o = -R_f \left(\frac{V_{s1}}{R_1} + \frac{V_{s2}}{R_2}\right)$
- **Subtractor:** $V_o = \frac{R_f}{R}(V_{s2} - V_{s1})$

These circuits are directly applicable in the implementation of neural network analog processing blocks.

9.6 Relevance to Neural Networks

Amplifiers and op-amps form the foundational components for analog neural networks, enabling signal scaling, weighted addition, and difference calculation. Understanding their characteristics and configurations allows for effective design of analog neuron circuits with precise gain and feedback control.