# Gesture Recognition using Neural Network

Gesture recognition is a vital technology with applications spanning human-computer interaction, virtual reality, robotics, and more. This project aims to develop a gesture recognition system using neural networks, a powerful machine learning technique capable of learning complex patterns from data.

**Model used**

In this project, we employed a neural network-based approach for gesture recognition. Neural networks, inspired by the human brain, consist of interconnected layers of neurons that can learn to perform tasks by adjusting their parameters through the process of training. Specifically, we used a feedforward neural network architecture, commonly implemented using the TensorFlow and Keras libraries in Python.

**Code Explanation**

Data Loading and Preprocessing:

- We start by loading the dataset from a CSV file containing sensor readings from wearable devices capturing hand gestures.
- The dataset is pre-processed to prepare it for training the neural network. This involves splitting the data into features (input) and target labels (output) and encoding categorical labels into numerical values.
- Input features are normalized to ensure uniform scale across different features, which helps improve the convergence of the neural network during training.
- The pre-processed data is then split into training and testing sets using the train_test_split function from scikit-learn.

**Model Building:**

- We define a neural network model using the Sequential API provided by Keras. This allows us to create a linear stack of layers for building the model architecture.
- The model consists of several dense (fully connected) layers, each followed by a Rectified Linear Unit (ReLU) activation function to introduce non-linearity into the model.
- Dropout layers are added to prevent overfitting by randomly dropping a fraction of input units during training.
- The output layer uses a softmax activation function to produce probability distributions over the classes, making it suitable for multi-class classification tasks.

In this project, we used a feedforward neural network (FNN) model for gesture recognition. Specifically, we employed a simple architecture consisting of multiple dense (fully connected) layers. Here's a breakdown of the neural network architecture we used:

**Input Layer:**

  - The input layer accepts the features (sensor readings) as input to the neural network.
  - It has a number of neurons equal to the number of features in the input data.

- We used the ReLU (Rectified Linear Unit) activation function in the input layer to introduce non-linearity into the model.

**Hidden Layers:**

- We added multiple hidden layers to capture complex patterns in the data.
- Each hidden layer consists of a dense (fully connected) layer with a specified number of neurons.
- We used the ReLU activation function in the hidden layers to introduce non-linearity and enable the model to learn complex representations of the input data.
- Dropout layers were added after each hidden layer to prevent overfitting by randomly dropping a fraction of input units during training.

**Output Layer**

- The output layer is the final layer of the neural network, responsible for producing the model's predictions.
- Since we're performing multi-class classification (recognizing different hand gestures), the output layer consists of multiple neurons, each corresponding to a class label.
- We used the softmax activation function in the output layer to convert the raw output values into probability distributions over the classes, allowing us to interpret the model's predictions as probabilities.

Overall, the neural network model used in this project is a simple feedforward architecture with multiple hidden layers, ReLU activation functions, and dropout regularization. This type of architecture is commonly used for various machine learning tasks, including classification problems like gesture recognition.

**Model Compilation and Training:**

- We compile the model using the Adam optimizer, a popular optimization algorithm known for its efficiency and effectiveness in training neural networks.
- The loss function is specified as sparse categorical cross-entropy, suitable for multi-class classification problems with integer-encoded labels.
- During training, we monitor the model's performance using accuracy as the evaluation metric.
- The model is trained on the training data for a specified number of epochs (iterations over the entire dataset) with a batch size of 64.
- A portion of the training data is used for validation to monitor the model's performance on unseen data during training.

**Evaluation and Visualization:**

- After training the model, we evaluate its performance on the test dataset to assess how well it generalizes to unseen data.
- We visualize the training progress by plotting the accuracy and validation accuracy over epochs using matplotlib.pyplot

In conclusion, we have successfully developed a gesture recognition system using neural networks. By leveraging the power of deep learning, we have created a model capable of accurately classifying hand gestures based on sensor data. This project demonstrates the effectiveness of neural networks in solving complex pattern recognition tasks and opens up opportunities for further research and applications in gesture recognition technology.

**Future Work**

While the current model shows promising results, there are several avenues for future exploration and improvement:

- Experiment with different neural network architectures, such as convolutional neural networks (CNNs), which are well-suited for processing spatial data like images.
- Fine-tune hyperparameters such as learning rate, number of layers, and dropout rates to optimize model performance.
- Explore additional preprocessing techniques and feature engineering methods to further enhance model accuracy.
- Deploy the trained model in real-world applications, such as gesture-controlled interfaces or robotics systems, and evaluate its performance in practical scenarios.

# Execution Code

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np

# Load the dataset from CSV
data = pd.read_csv("/content/drive/MyDrive/dataset.csv")

# Split dataset into features (X) and target (y)
X = data.drop(columns=['GESTURE'])
y = data['GESTURE']

# Encode categorical target variable
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# Normalize input features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled,
y_encoded, test_size=0.2, random_state=42)

# Building the Model
model = keras.Sequential([
    layers.Dense(256, activation='relu',
input_shape=(X_train.shape[1],)),
    layers.Dropout(0.5),  # Adding dropout for regularization
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),  # Adding dropout for regularization
    layers.Dense(64, activation='relu'),
    layers.Dense(len(label_encoder.classes_), activation='softmax')  #
Output layer with softmax activation for multi-class classification
])

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',  # Using sparse
categorical cross-entropy for multi-class classification
              metrics=['accuracy'])

# Training the Model
history = model.fit(X_train, y_train, epochs=50, batch_size=64,
validation_split=0.2)
```

```
# Smoothing the training curves using exponential moving averages (EMA)
def smooth_curve(points, factor=0.8):
    smoothed_points = []
    for point in points:
        if smoothed_points:
            previous = smoothed_points[-1]
            smoothed_points.append(previous * factor + point * (1 -
factor))
        else:
            smoothed_points.append(point)
    return smoothed_points

smoothed_accuracy = smooth_curve(history.history['accuracy'])
smoothed_val_accuracy = smooth_curve(history.history['val_accuracy'])

# Plotting Smoothed Accuracy Graph
plt.plot(range(1, len(smoothed_accuracy) + 1), smoothed_accuracy,
label='accuracy')
plt.plot(range(1, len(smoothed_val_accuracy) + 1),
smoothed_val_accuracy, label='val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.show()

# Evaluating the Model on Test Data
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f'Test Loss: {test_loss}')
print(f'Test Accuracy: {test_accuracy}')
```

# Epoch Running Rate

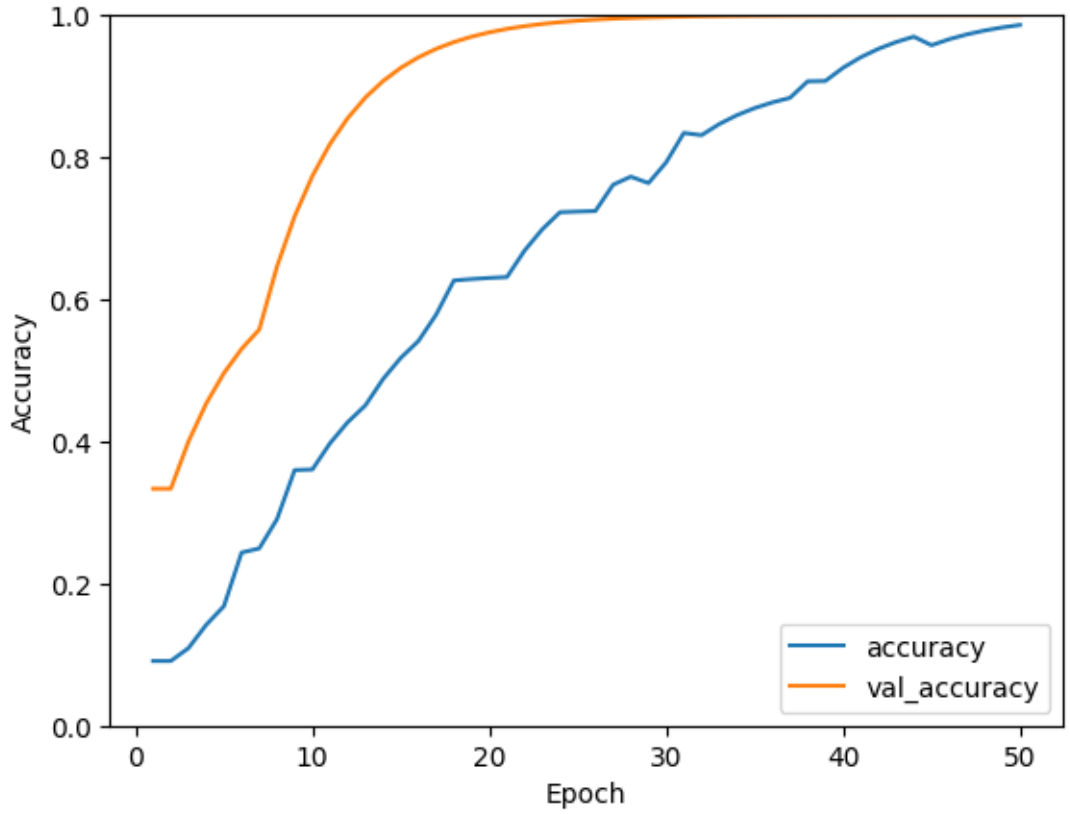
```
Epoch 1/50
1/1 [==============================] - 1s 1s/step - loss: 1.9578 -
accuracy: 0.0909 - val_loss: 1.6631 - val_accuracy: 0.3333
Epoch 2/50
1/1 [==============================] - 0s 62ms/step - loss: 1.9870 -
accuracy: 0.0909 - val_loss: 1.6395 - val_accuracy: 0.3333
Epoch 3/50
1/1 [==============================] - 0s 39ms/step - loss: 1.7929 -
accuracy: 0.1818 - val_loss: 1.6154 - val_accuracy: 0.6667
Epoch 4/50
1/1 [==============================] - 0s 40ms/step - loss: 1.6785 -
accuracy: 0.2727 - val_loss: 1.5895 - val_accuracy: 0.6667
Epoch 5/50
1/1 [==============================] - 0s 37ms/step - loss: 1.6103 -
accuracy: 0.2727 - val_loss: 1.5644 - val_accuracy: 0.6667
Epoch 6/50
1/1 [==============================] - 0s 39ms/step - loss: 1.4508 -
accuracy: 0.5455 - val_loss: 1.5383 - val_accuracy: 0.6667
Epoch 7/50
1/1 [==============================] - 0s 39ms/step - loss: 1.5730 -
accuracy: 0.2727 - val_loss: 1.5136 - val_accuracy: 0.6667
Epoch 8/50
1/1 [==============================] - 0s 42ms/step - loss: 1.5310 -
accuracy: 0.4545 - val_loss: 1.4872 - val_accuracy: 1.0000
Epoch 9/50
1/1 [==============================] - 0s 39ms/step - loss: 1.5315 -
accuracy: 0.6364 - val_loss: 1.4585 - val_accuracy: 1.0000
Epoch 10/50
1/1 [==============================] - 0s 40ms/step - loss: 1.4491 -
accuracy: 0.3636 - val_loss: 1.4287 - val_accuracy: 1.0000
Epoch 11/50
1/1 [==============================] - 0s 38ms/step - loss: 1.2397 -
accuracy: 0.5455 - val_loss: 1.3984 - val_accuracy: 1.0000
Epoch 12/50
1/1 [==============================] - 0s 38ms/step - loss: 1.2941 -
accuracy: 0.5455 - val_loss: 1.3665 - val_accuracy: 1.0000
Epoch 13/50
1/1 [==============================] - 0s 56ms/step - loss: 1.1494 -
accuracy: 0.5455 - val_loss: 1.3344 - val_accuracy: 1.0000
Epoch 14/50
1/1 [==============================] - 0s 41ms/step - loss: 1.0972 -
accuracy: 0.6364 - val_loss: 1.3024 - val_accuracy: 1.0000
Epoch 15/50
1/1 [==============================] - 0s 42ms/step - loss: 1.0570 -
accuracy: 0.6364 - val_loss: 1.2717 - val_accuracy: 1.0000
Epoch 16/50
1/1 [==============================] - 0s 43ms/step - loss: 1.0931 -
accuracy: 0.6364 - val_loss: 1.2419 - val_accuracy: 1.0000
Epoch 17/50
1/1 [==============================] - 0s 38ms/step - loss: 0.8441 -
accuracy: 0.7273 - val_loss: 1.2125 - val_accuracy: 1.0000
Epoch 18/50
1/1 [==============================] - 0s 37ms/step - loss: 0.9656 -
accuracy: 0.8182 - val_loss: 1.1834 - val_accuracy: 1.0000
Epoch 19/50
```

```
1/1 [==============================] - 0s 39ms/step - loss: 0.8989 -
accuracy: 0.6364 - val_loss: 1.1522 - val_accuracy: 1.0000
Epoch 20/50
1/1 [==============================] - 0s 39ms/step - loss: 0.8679 -
accuracy: 0.6364 - val_loss: 1.1190 - val_accuracy: 1.0000
Epoch 21/50
1/1 [==============================] - 0s 40ms/step - loss: 0.9726 -
accuracy: 0.6364 - val_loss: 1.0847 - val_accuracy: 1.0000
Epoch 22/50
1/1 [==============================] - 0s 38ms/step - loss: 0.6782 -
accuracy: 0.8182 - val_loss: 1.0507 - val_accuracy: 1.0000
Epoch 23/50
1/1 [==============================] - 0s 37ms/step - loss: 0.8522 -
accuracy: 0.8182 - val_loss: 1.0160 - val_accuracy: 1.0000
Epoch 24/50
1/1 [==============================] - 0s 56ms/step - loss: 0.6650 -
accuracy: 0.8182 - val_loss: 0.9806 - val_accuracy: 1.0000
Epoch 25/50
1/1 [==============================] - 0s 49ms/step - loss: 0.8491 -
accuracy: 0.7273 - val_loss: 0.9458 - val_accuracy: 1.0000
Epoch 26/50
1/1 [==============================] - 0s 37ms/step - loss: 0.6908 -
accuracy: 0.7273 - val_loss: 0.9098 - val_accuracy: 1.0000
Epoch 27/50
1/1 [==============================] - 0s 37ms/step - loss: 0.7644 -
accuracy: 0.9091 - val_loss: 0.8751 - val_accuracy: 1.0000
Epoch 28/50
1/1 [==============================] - 0s 40ms/step - loss: 0.6049 -
accuracy: 0.8182 - val_loss: 0.8410 - val_accuracy: 1.0000
Epoch 29/50
1/1 [==============================] - 0s 39ms/step - loss: 0.7421 -
accuracy: 0.7273 - val_loss: 0.8065 - val_accuracy: 1.0000
Epoch 30/50
1/1 [==============================] - 0s 38ms/step - loss: 0.5314 -
accuracy: 0.9091 - val_loss: 0.7730 - val_accuracy: 1.0000
Epoch 31/50
1/1 [==============================] - 0s 37ms/step - loss: 0.4172 -
accuracy: 1.0000 - val_loss: 0.7401 - val_accuracy: 1.0000
Epoch 32/50
1/1 [==============================] - 0s 37ms/step - loss: 0.5964 -
accuracy: 0.8182 - val_loss: 0.7069 - val_accuracy: 1.0000
Epoch 33/50
1/1 [==============================] - 0s 58ms/step - loss: 0.5381 -
accuracy: 0.9091 - val_loss: 0.6755 - val_accuracy: 1.0000
Epoch 34/50
1/1 [==============================] - 0s 37ms/step - loss: 0.4044 -
accuracy: 0.9091 - val_loss: 0.6445 - val_accuracy: 1.0000
Epoch 35/50
1/1 [==============================] - 0s 54ms/step - loss: 0.3823 -
accuracy: 0.9091 - val_loss: 0.6147 - val_accuracy: 1.0000
Epoch 36/50
1/1 [==============================] - 0s 39ms/step - loss: 0.4358 -
accuracy: 0.9091 - val_loss: 0.5850 - val_accuracy: 1.0000
Epoch 37/50
1/1 [==============================] - 0s 37ms/step - loss: 0.3298 -
accuracy: 0.9091 - val_loss: 0.5569 - val_accuracy: 1.0000
Epoch 38/50
1/1 [==============================] - 0s 40ms/step - loss: 0.2120 -
accuracy: 1.0000 - val_loss: 0.5300 - val_accuracy: 1.0000
Epoch 39/50
```

```
1/1 [==============================] - 0s 40ms/step - loss: 0.3963 -
accuracy: 0.9091 - val_loss: 0.5041 - val_accuracy: 1.0000
Epoch 40/50
1/1 [==============================] - 0s 39ms/step - loss: 0.2639 -
accuracy: 1.0000 - val_loss: 0.4797 - val_accuracy: 1.0000
Epoch 41/50
1/1 [==============================] - 0s 40ms/step - loss: 0.3008 -
accuracy: 1.0000 - val_loss: 0.4564 - val_accuracy: 1.0000
Epoch 42/50
1/1 [==============================] - 0s 41ms/step - loss: 0.2477 -
accuracy: 1.0000 - val_loss: 0.4338 - val_accuracy: 1.0000
Epoch 43/50
1/1 [==============================] - 0s 39ms/step - loss: 0.1529 -
accuracy: 1.0000 - val_loss: 0.4120 - val_accuracy: 1.0000
Epoch 44/50
1/1 [==============================] - 0s 38ms/step - loss: 0.2814 -
accuracy: 1.0000 - val_loss: 0.3908 - val_accuracy: 1.0000
Epoch 45/50
1/1 [==============================] - 0s 42ms/step - loss: 0.2698 -
accuracy: 0.9091 - val_loss: 0.3718 - val_accuracy: 1.0000
Epoch 46/50
1/1 [==============================] - 0s 40ms/step - loss: 0.1126 -
accuracy: 1.0000 - val_loss: 0.3538 - val_accuracy: 1.0000
Epoch 47/50
1/1 [==============================] - 0s 42ms/step - loss: 0.2244 -
accuracy: 1.0000 - val_loss: 0.3348 - val_accuracy: 1.0000
Epoch 48/50
1/1 [==============================] - 0s 47ms/step - loss: 0.1770 -
accuracy: 1.0000 - val_loss: 0.3173 - val_accuracy: 1.0000
Epoch 49/50
1/1 [==============================] - 0s 41ms/step - loss: 0.1410 -
accuracy: 1.0000 - val_loss: 0.3005 - val_accuracy: 1.0000
Epoch 50/50
1/1 [==============================] - 0s 38ms/step - loss: 0.1444 -
accuracy: 1.0000 - val_loss: 0.2850 - val_accuracy: 1.0000
```

# Output

**Accuracy Graph:**




**Accuracy Obtained:**

```
Test Loss: 0.7734860181808472
Test Accuracy: 0.5
```