## Objective

This document evaluates three relevant research papers and proposes the optimal approach for implementation.

Overview

**Research Paper #1:** <https://www.mdpi.com/1424-8220/20/2/376>

**Title:** Trajectory-Based Air Writing Recognition Using Deep Neural Network and Depth Sensor

**Key Takeaways:**

* **Setup and Data Collection:** Used depth sensors for capturing spatial trajectory sequences.
* **Preprocessing Techniques:**
  + **Nearest Neighbor Normalization:** Smoothed trajectories by adjusting six neighboring points, reducing noise effectively. This was more accurate than the root point normalization.
  + **Root Point Normalization:** Standardized trajectories to a common origin for consistency.
* **Modeling Approach:**
  + **LSTM for Digit Recognition:** Captured sequential dependencies in trajectories and was specifically used for recognizing digits written in the air. Compared to CNN, LSTM excelled at capturing long-term dependencies inherent in sequential data.
  + **CNN:** Extracted spatial features but proved less effective for digit recognition due to its inability to fully model temporal relationships in trajectory data.
* **Findings:** LSTM performed better due to its ability to handle long-term dependencies in trajectory data.

**Relevance to Our Project:**

* The trajectory smoothing and LSTM-based modeling approaches can be adapted for LED gloves, leveraging the sequential nature of movement data. LSTM’s success in digit recognition can guide our approach to text inference.

**Research Paper #2:** <https://dl.acm.org/doi/abs/10.1145/3474085.3475694>

**Title:** Air-Text: Air-Writing and Recognition System

**Github Repo:** <https://github.com/sklee2014/Air-Text?utm_source=chatgpt.com>

**Key Takeaways:**

* **Modules:**
  + **Air-Writing Module:** Tracks fingertip locations from RGB video input and converts trajectories into binary images.
  + **Text-Recognition Module:** Utilizes a pre-trained model (TPS-ResNet-BiLSTM-Attn) for text prediction.
* **Modeling Details:**
  + **TPS-ResNet-BiLSTM-Attn:** Combines Thin Plate Spline (TPS) for alignment, ResNet for feature extraction, BiLSTM for sequence modeling, and Attention for accurate text recognition.

**Relevance to Our Project:**

* The system’s fingertip tracking can be adapted to LED brightness localization.
* The TPS-ResNet-BiLSTM-Attn model provides a strong foundation for trajectory-to-text conversion.

**Research Paper #3:** <https://ieeexplore.ieee.org/abstract/document/9257775>

**Title:** Air-Writing Recognition using Deep Convolutional and Recurrent Neural Network Architectures

**Github Repo:** <https://github.com/kosmasK/air-writing-recognition?utm_source=chatgpt.com>

**Key Takeaways:**

* **Objective:** This study focuses on recognizing air-written digits (0-9) using hand movements tracked in 3D space by a Leap Motion Controller (LMC). Both time-series (dynamic) and image-based (static) approaches were evaluated.
* **Dynamic Models:**
  + **LSTM:** Performed best for recognizing the sequential patterns in air-written digits, with high accuracy.
  + **BLSTM:** Slightly less accurate than LSTM.
  + **CNN-LSTM:** Combined spatial feature extraction with sequential modeling but was less effective than LSTM alone.
  + **TCN:** Fast and efficient but less accurate than LSTM.
* **Static Models:**
  + **CNN:** Processed binary images of digit trails with good accuracy but was less effective than LSTM for sequential data.
  + **TCN-Static:** Performed the worst due to loss of spatial details during conversion.
* **Combined Model:** Combining LSTM and CNN didn’t improve performance and slightly reduced accuracy compared to LSTM alone.

**Key Findings:**

* LSTM excelled at handling time-series data, making it the best choice for air-writing recognition.
* Static models like CNN worked well for image-based data but were less effective for sequential gestures.

**Relevance to Our Project:**

* This paper highlights that LSTM is ideal for recognizing the sequential patterns in air-written digits, which directly applies to tracking LED glove trajectories.
* The findings reinforce the importance of using dynamic models like LSTM for accurate and efficient recognition of air-writing gestures.
* Insights from combining spatial and sequential features can help refine our approach for trajectory-to-text conversion.

## Proposed Approach for Implementation

* **Preprocessing:**
  + Apply Nearest Neighbor Normalization to smooth trajectories.
* **Modeling Approach:**
  + **Trajectory Tracking:** Implement fingertip tracking using YOLO for real-time LED localization.
  + **Text Recognition:** Utilize a combination of LSTM for sequential digit recognition and TPS-ResNet-BiLSTM-Attn for more complex word inference.

Step 1: Define the Dataset Format

The dataset format should be as follows:

* **Images:** Grayscale 28x28 PNG images of each air-written digit.
* **Labels:** A CSV file mapping each image filename to its corresponding digit label.

Step 2: Collect Data from the Mobile App

* When you write a digit in the air, take a screenshot of the final written stroke.
* Save the screenshot as a PNG file.
* Store all the images in a folder.
* Resize the images to 28x28 Pixels before training.
  + Run this script to resize all the images:

import os

import cv2

# Set dataset path

# dataset\_path = enter the dataset path

# output\_path = enter where you want the output to be

# Create output directory if it doesn't exist

if not os.path.exists(output\_path):

os.makedirs(output\_path)

# Loop through all images

for img\_file in os.listdir(dataset\_path):

if img\_file.endswith(('.png', '.jpg', '.jpeg')): # Check if it's an image

img\_path = os.path.join(dataset\_path, img\_file)

# Read image

img = cv2.imread(img\_path, cv2.IMREAD\_GRAYSCALE) # Convert to grayscale

img = cv2.resize(img, (28, 28)) # Resize to 28x28

# Save the resized image

cv2.imwrite(os.path.join(output\_path, img\_file), img)

print("All images have been resized to 28x28 and saved in:", output\_path)

* **After resizing it:**
  + Save the filename and corresponding digit label in a CSV file (labels.csv).
  + Example of CSV file:

image,label

00001.png,3

00002.png,7

00003.png,0

### **Instructions to Train the Model on the New Dataset**

Now that you've collected, resized, and labeled the dataset, it's time to train the model using the new air-writing dataset.

### **Step 1: Move the Dataset to the Training Folder**

Ensure that your dataset is structured properly before training:

bash

CopyEdit

airwriting\_dataset/

├── images/ # Folder containing 28x28 grayscale images

│ ├── 00001.png

│ ├── 00002.png

│ └── ...

├── labels.csv # CSV file mapping filenames to digits

### **Step 2: Modify the Training Script**

Update the training script to load the new dataset.

#### **Modify train\_model.py**

Replace the dataset loading section with this:

python

CopyEdit

import torch

import torchvision

import torchvision.transforms as transforms

from torch.utils.data import DataLoader, Dataset

import pandas as pd

import os

from PIL import Image

# Custom dataset class to load images and labels from CSV

class AirWritingDataset(Dataset):

def \_\_init\_\_(self, image\_folder, labels\_file, transform=None):

self.image\_folder = image\_folder

self.labels = pd.read\_csv(labels\_file)

self.transform = transform

def \_\_len\_\_(self):

return len(self.labels)

def \_\_getitem\_\_(self, idx):

img\_name = os.path.join(self.image\_folder, self.labels.iloc[idx, 0])

image = Image.open(img\_name).convert("L") # Convert to grayscale

label = int(self.labels.iloc[idx, 1])

if self.transform:

image = self.transform(image)

return image, label

# Define transformations (ensure images are normalized)

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.5,), (0.5,))

])

# Load dataset

dataset\_path = "path/to/airwriting\_dataset/images"

labels\_path = "path/to/airwriting\_dataset/labels.csv"

train\_dataset = AirWritingDataset(dataset\_path, labels\_path, transform=transform)

train\_loader = DataLoader(train\_dataset, batch\_size=64, shuffle=True)

### **Step 3: Train the Model**

Now, train the model using the new dataset.

bash

CopyEdit

python train\_model.py

### **Step 4: Monitor Training**

While training, check the console output for loss and accuracy updates.

If the dataset is small, consider using **data augmentation** to increase training diversity.

### **Step 5: Save the Trained Model**

After training, the model should be saved for later testing:

python

CopyEdit

torch.save(model.state\_dict(), "digit\_recognizer\_airwriting.pth")

This saved file can now be used for **real-time inference** in the mobile application.

### **Instructions to Test the Trained Model on the New Dataset**

Once the model is trained, we need to evaluate its accuracy on new air-written digits.

### **Step 1: Prepare the Testing Script**

Create a new script **test\_model.py** to evaluate the trained model.

#### **Create test\_model.py**

python

CopyEdit

import torch

import torchvision.transforms as transforms

import pandas as pd

import os

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

from torch.utils.data import DataLoader, Dataset

# Define the trained model

class DigitRecognizer(torch.nn.Module):

def \_\_init\_\_(self):

super(DigitRecognizer, self).\_\_init\_\_()

self.conv1 = torch.nn.Conv2d(1, 16, kernel\_size=5, stride=1, padding=2)

self.conv2 = torch.nn.Conv2d(16, 32, kernel\_size=5, stride=1, padding=2)

self.fc1 = torch.nn.Linear(7 \* 7 \* 32, 128)

self.fc2 = torch.nn.Linear(128, 10)

def forward(self, x):

x = torch.relu(self.conv1(x))

x = torch.max\_pool2d(x, 2)

x = torch.relu(self.conv2(x))

x = torch.max\_pool2d(x, 2)

x = x.view(-1, 7 \* 7 \* 32)

x = torch.relu(self.fc1(x))

x = self.fc2(x)

return x

# Custom dataset class for testing

class AirWritingDataset(Dataset):

def \_\_init\_\_(self, image\_folder, labels\_file, transform=None):

self.image\_folder = image\_folder

self.labels = pd.read\_csv(labels\_file)

self.transform = transform

def \_\_len\_\_(self):

return len(self.labels)

def \_\_getitem\_\_(self, idx):

img\_name = os.path.join(self.image\_folder, self.labels.iloc[idx, 0])

image = Image.open(img\_name).convert("L") # Convert to grayscale

label = int(self.labels.iloc[idx, 1])

if self.transform:

image = self.transform(image)

return image, label, img\_name # Returning filename for visualization

# Define transformations

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.5,), (0.5,))

])

# Load dataset

dataset\_path = "path/to/airwriting\_dataset/images"

labels\_path = "path/to/airwriting\_dataset/labels.csv"

test\_dataset = AirWritingDataset(dataset\_path, labels\_path, transform=transform)

test\_loader = DataLoader(test\_dataset, batch\_size=1, shuffle=True)

# Load the trained model

device = torch.device("cpu")

model = DigitRecognizer().to(device)

model.load\_state\_dict(torch.load("digit\_recognizer\_airwriting.pth", map\_location=device))

model.eval()

# Function to evaluate the model

def test\_model(model, test\_loader):

model.eval()

correct = 0

total = 0

sample\_images = []

sample\_labels = []

sample\_predictions = []

with torch.no\_grad():

for images, labels, img\_names in test\_loader:

images, labels = images.to(device), labels.to(device)

outputs = model(images)

predicted = torch.argmax(outputs, dim=1)

correct += (predicted == labels).sum().item()

total += labels.size(0)

# Store a few sample predictions for visualization

if len(sample\_images) < 10:

sample\_images.append(img\_names[0]) # Store image filename

sample\_labels.append(labels.item())

sample\_predictions.append(predicted.item())

accuracy = 100 \* correct / total

print(f"\n✅ Test Accuracy: {accuracy:.2f}%\n")

# Display sample predictions

print("🔍 Sample Predictions:")

for i in range(len(sample\_images)):

print(f"🖼️ Image {i+1}: {sample\_images[i]} | True Label: {sample\_labels[i]}, Predicted: {sample\_predictions[i]}")

return sample\_images, sample\_labels, sample\_predictions

# Run the test function

sample\_images, sample\_labels, sample\_predictions = test\_model(model, test\_loader)

# Display the first few test images with predictions

fig, axes = plt.subplots(2, 5, figsize=(12, 5))

for i, ax in enumerate(axes.flat):

if i < len(sample\_images):

img = Image.open(sample\_images[i]) # Load image for visualization

ax.imshow(img, cmap='gray')

ax.set\_title(f"True: {sample\_labels[i]}, Pred: {sample\_predictions[i]}")

ax.axis("off")

plt.show()

### **Step 2: Run the Testing Script**

Once the script is ready, execute the following command to test the model:

bash

CopyEdit

python test\_model.py