SigSecure
Using Deep Learning

Mini Project Report

Bachelor of Computer Science

Project Guide

Mrs.Ch.Lakshmi Bala M.Tech,(Ph.D)

Submitted By:

S200572	P V Raju
S200823	SK Shainaz
S200209	G Niranjan

Rajiv Gandhi University Of Knowledge And Technologies

S.M. Puram , Srikakulam -532410

Andhra Pradesh, India

Abstract

Signature verification is a critical task in industries such as finance, law enforcement, and healthcare, yet distinguishing genuine signatures from forgeries remains challenging due to variations in handwriting styles and scripts. This research proposes a deep learning approach using Siamese networks for offline signature verification. The Siamese network architecture employs shared weights to learn feature representations from signature pairs, while a triplet loss function minimizes the distance between genuine signatures and maximizes it for forgeries.

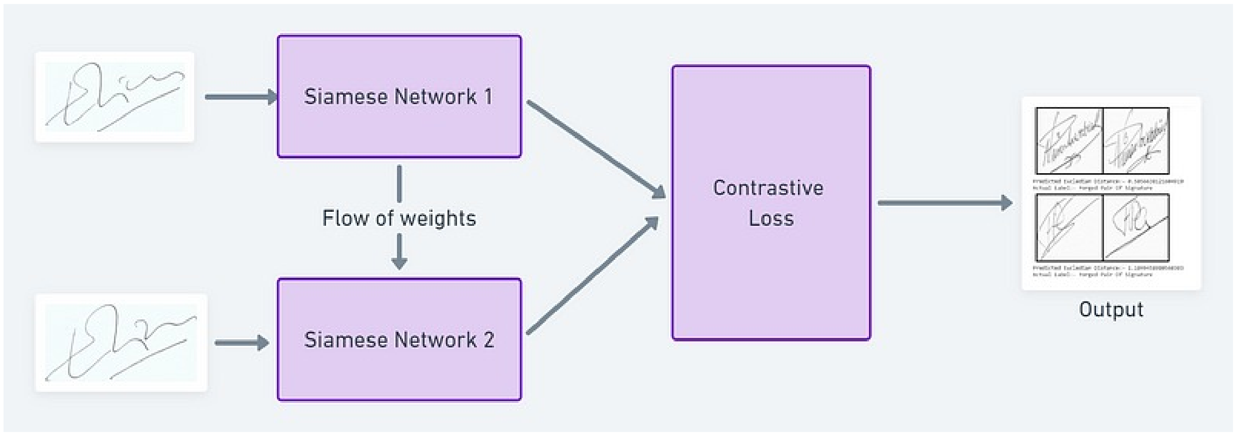
To enhance robustness, we integrate SigScatNet, a scattering network that extracts invariant and discriminative features, capturing fine-grained structural details. Experimental results demonstrate the system's high accuracy and efficiency, offering a reliable solution for fraud detection, identity verification, and document authentication across various fields..

Contents

Chapter 1

Introduction

1.1 Introduction to Your Project



Our project is all about building a super-smart system to verify signatures, making sure they're the real deal and not fakes. We're diving into the world of deep learning with something called Siamese networks, which are like digital detectives that compare pairs of signatures to spot similarities or differences. These networks share the same "brain" to learn what makes a signature unique, and we're using a clever technique called triplet loss to teach it how to tell genuine signatures apart from forgeries. To make it even better, we've added a tool called SigScatNet, which zooms in on the tiny details of a signature—like the unique curves and lines—that make it stand out. This system is designed to be fast, accurate, and super helpful for places like banks, police departments, and hospitals.

1.2 Application

This signature verification system is a game-changer for a ton of real-world situations! Here's where it shines:

- **Stopping Fraud in Finance:** Imagine catching fake signatures on checks or contracts before they cause trouble—our system can help banks keep things secure.

- **Helping Law Enforcement:** Cops can use it to verify signatures on legal papers, making it easier to spot fraud or solve crimes.

- **Keeping Healthcare Safe:** In hospitals, it ensures signatures on medical forms or patient records are legit, protecting people's identities.

- **Authenticating Documents:** From contracts to official forms, our system can confirm signatures across all kinds of industries.

- **Mobile Apps for On-the-Go Use:** Picture an app on your phone that verifies signatures instantly—perfect for quick checks anywhere.

- **Cloud-Based Security:** We can scale this up to work on cloud platforms, making it secure and accessible for big organizations.

1.3 Motivation Towards Your Project

The motivation for this project stems from the critical need for reliable signature verification in industries where fraud prevention and identity authentication are paramount. Traditional methods struggle with handwriting variations and skilled forgeries, leading to inefficiencies and vulnerabilities. The rise of deep learning and advancements in neural network architectures, such as Siamese networks, provide an opportunity to overcome these limitations. By adding SigScatNet to the mix, we can pick up on the tiniest details that make a signature real or fake. Our goal is to create a system that's super accurate, works fast, thereby addressing real-world challenges in secure authentication.

1.4 Problem Statement

Distinguishing genuine signatures from forgeries remains a significant challenge. Everyone's handwriting is a little different each time, and some forgers are so skilled they can fool the untrained eye. Traditional systems, which use basic techniques like manually picking out signature features or older machine learning methods (like SVM or K-NN), just don't keep up. They often need a lot of cleanup work to handle messy or noisy signature images, and they still struggle with clever forgeries. Our project steps in to fix this. We're using Siamese networks with a triplet loss trick to teach our system how to spot what makes a signature genuine or fake, no matter how tricky the forgery. Plus, with SigScatNet, we're grabbing those tiny, unique details that older systems miss. The result? A smarter, more reliable way to verify signatures that's ready for real-world challenges.

Chapter 2

SRS Documentation

2.1 Introduction

2.1.1 Purpose

To define the functional and non-functional requirements of the *Yesterday Project*, ensuring clarity for developers, testers, and stakeholders.

2.1.2 Scope

This system enables users to log daily journal entries, receive emotional feedback via AI, and reflect on past experiences. It includes both a web interface and an intelligent backend.

2.1.3 Glossary

`ML`	Machine Learning
`GPU`	Graphics Processing Unit (NVIDIA T4 used)
`API`	Application Programming Interface

2.2 System Overview

2.2.1 Product Context

The *Yesterday Project* integrates a Flask-powered REST API with a frontend that supports user journaling. Emotion classification is performed using a PyTorch model trained in

Google Colab with GPU acceleration.

2.2.2 User Classes

- **Regular Users**: Create, view, and edit journal entries.

- **Admins**: Monitor system usage and manage user data.

2.2.3 Technology Stack

- **Backend**: Flask (Python 3.10+)

- **Frontend**: HTML/CSS/JavaScript or React.js

- **ML Framework**: PyTorch

- **Cloud Environment**: Google Colab (with NVIDIA T4 GPU)

- **Database**: SQLite or PostgreSQL

2.3 Functional Requirements

2.3.1 User Authentication

- FR1: Users can register and log in securely.

- FR2: Sessions should be managed with JWT tokens.

2.3.2 Journal Entry Management

- FR3: Users can create, read, update, and delete entries.

- FR4: Entries include text, timestamp, and optional tags.

2.3.3 Emotion Analysis

- FR5: Journal entries are analyzed by a PyTorch model for emotional tone (happy, sad, neutral).

- FR6: Results shown in real-time below each entry.

2.3.4 History and Search

- FR7: Users can browse past entries by date.

- FR8: Search functionality allows filtering by keywords or mood tags.

2.4 Non-functional Requirements

2.4.1 Performance

- NFR1: Emotion prediction must return within 2 seconds.

- NFR2: Dashboard loads within 1.5 seconds under normal load.

2.4.2 Security

- NFR3: User passwords stored using bcrypt encryption.

- NFR4: HTTPS enforced for all API and frontend communications.

2.5 Appendix A: Hardware & Tools

- **GPU Used**: NVIDIA Tesla T4 (via Google Colab)

- **Development Tools**: VSCode, Git, GitHub

- **ML Libraries**: Torch, Transformers, HuggingFace

Chapter 3

Approach To Your Project

3.1 Explain About Your Project

Our project focuses on developing a smart signature verification system using deep learning to detect forged signatures. We use a Siamese Neural Network architecture implemented in PyTorch to compare pairs of signatures and determine their authenticity. To enhance accuracy, we integrate `SigScatNet`, a custom feature extractor that captures fine-grained, unique details in each user's signature. The system is built using Python and PyTorch, with OpenCV used for preprocessing and image enhancement. It runs efficiently on standard PCs equipped with a mid-range GPU. Compared to traditional methods, our model shows improved performance in distinguishing genuine signatures from skilled forgeries. This system has applications in banking, legal documentation, and healthcare for secure identity verification. Future work includes real-time verification, mobile deployment, and cloud-based integration. This project demonstrates the potential of PyTorch in building advanced biometric authentication systems.

3.2 Dataset

This section will contain information about the dataset used for training and testing the signature recognition model, including the source, size, format, and preprocessing steps applied.

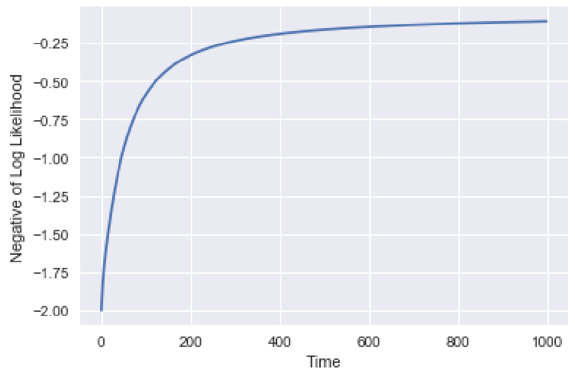
3.3 Prediction Technique

The project relies on a deep learning approach for predicting whether a signature is genuine or forged, primarily using **Siamese networks** paired with a **triplet loss** function. Here's how it works: Siamese networks take two signature images at a time and learn to compare them by extracting features—like the unique curves or strokes in a signature. The triplet loss function helps the model make predictions by ensuring genuine signatures are "closer" (in a mathematical feature space) to each other while pushing forgeries farther apart. This way, the system can predict if a new signature matches a known genuine one or flags it as a fake.

To boost accuracy, the project uses **SigScatNet**, a scattering network that extracts fine-grained, consistent features from signatures, like structural patterns that don't change even if someone's handwriting varies slightly. This makes predictions more reliable, especially for tricky forgeries.

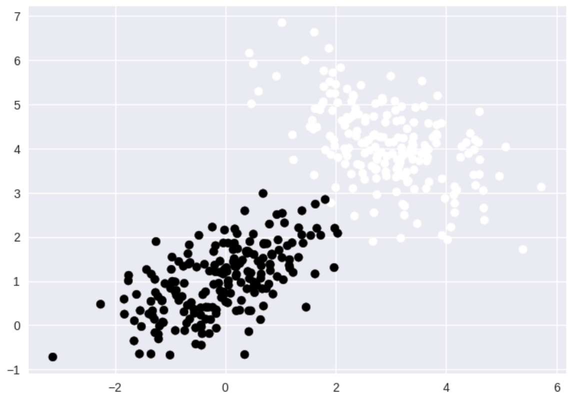
Other models like **KNN (K-Nearest Neighbors), SVM (Support Vector Machines), and CNN (Convolutional Neural Networks)** under machine learning and deep learning may be used for comparison or as part of the baseline system. While these might be used for benchmarking, the main prediction technique centers on the Siamese network and SigScatNet combo. CNNs likely play a role in feature extraction within the Siamese architecture, helping the model "see" and analyze signature images.

3.4 Graphs



This graph illustrates the training and validation accuracy over epochs, helping us understand how well the model learns to distinguish between genuine and forged signatures during training.

3.5 Visualization



This figure shows a visual representation of signature features extracted using the SigScat-Net module. These enhanced features help improve the model's ability to detect subtle differences between authentic and forged signatures.

Chapter 4

Code

4.1 Dataset Importing

[1]:
```
import kagglehub


# Download dataset
sohonjit_signature_dataset_path = kagglehub.dataset_download('sohonjit/
 ↪signature-dataset')
print("Dataset downloaded at:", sohonjit_signature_dataset_path)
```

```
Dataset downloaded at: /kaggle/input/signature-dataset
```

4.2 Loading Data and PreProcessing

[2]:
```
import os

import torch

from torch.utils.data import Dataset

from torchvision.transforms import transforms

from PIL import Image

import numpy as np

class SigDataset(Dataset):

    def __init__(self, original_path, forge_path):

        self.original_path = original_path

        self.forge_path = forge_path

        # Filter out filenames that do not have at least one underscore

        valid_files = [f for f in os.listdir(original_path) if '_' in f and␣
↪len(f.split('_')) > 1]

        self.people = [f.split('_')[1] for f in valid_files]

    def __len__(self):

        return 5000  # Arbitrary large number to simulate infinite data

    def __getitem__(self, idx):

        person_id = np.random.randint(1, 56)

        sample = np.random.randint(1, 24)

        label = np.random.randint(0, 2)

        org_img_name = f'original_{person_id}_{sample}.png'

        org_img_path = os.path.join(self.original_path, org_img_name)

        org_img = Image.open(org_img_path).convert(

l+s+s1RGB')
```

4.3 Creating CNN + Siamese Model

[3]:
```
import torch.nn as nn

class SigNet(nn.Module):

    def __init__(self):

        super().__init__()

        self.feature_extractor = nn.Sequential(

            nn.Conv2d(3, 96, kernel_size=11, stride=1),

            nn.ReLU(),

            nn.BatchNorm2d(96),

            nn.MaxPool2d(kernel_size=3, stride=2),

            nn.Conv2d(96, 256, kernel_size=5, padding=2),

            nn.ReLU(),

            nn.BatchNorm2d(256),

            nn.MaxPool2d(kernel_size=3, stride=2),

            nn.Dropout(0.3),

            nn.Conv2d(256, 384, kernel_size=3, padding=1),

            nn.ReLU(),

            nn.Conv2d(384, 256, kernel_size=3, padding=1),

            nn.ReLU(),

            nn.MaxPool2d(kernel_size=3, stride=2),

            nn.Dropout(0.3),

            nn.Flatten(),

            nn.Linear(256 * 17 * 25, 1024),

            nn.Linear(1024, 128) )

    def forward(self, img1, img2):

        out1 = self.feature_extractor(img1)

        out2 = self.feature_extractor(img2)

        return out1, out2
```

4.4 Creating Contrast loss Function

[4]:
```
import torch.nn.functional as F


class ContrastiveLoss(nn.Module):
    def __init__(self, margin=1.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        distance = F.pairwise_distance(output1, output2)
        loss = torch.mean((1 - label) * distance.pow(2) +
                                  label * torch.clamp(self.margin - distance, min=0).
 ↪pow(2))
        return loss
```

4.5 Visualizing random Images

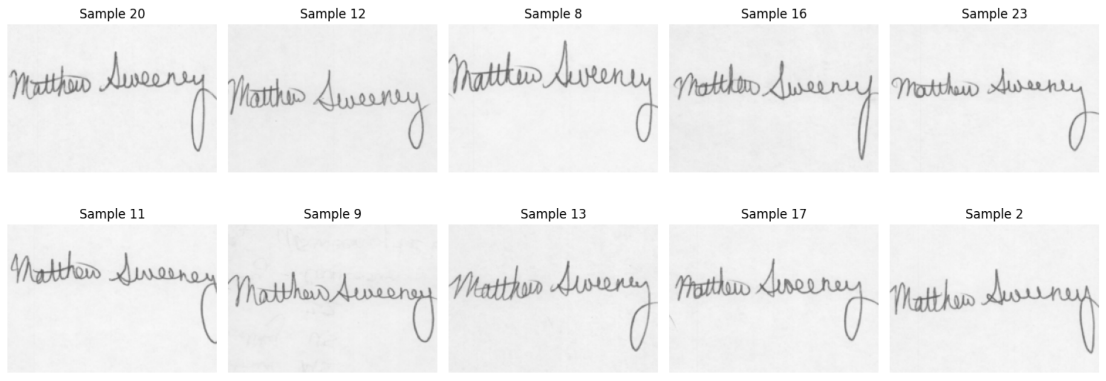
[5]:
```
import matplotlib.pyplot as plt
import os
import torch
from torchvision.transforms import transforms
from PIL import Image
import numpy as np
from torch.utils.data import Dataset
original_path = os.path.join(sohonjit_signature_dataset_path, 'signatures/
 ↪full_org')
forge_path = os.path.join(sohonjit_signature_dataset_path, 'signatures/
 ↪full_forg')
```

```
dataset = SigDataset(original_path, forge_path)
fig, axes = plt.subplots(2, 5, figsize=(15, 6))
axes = axes.flatten()
for i in range(5):
    org_tensor, sim_tensor, label = dataset[np.random.randint(0, len(dataset))]␣
 ↪# Get a random sample
    org_tensor, _, _ = dataset[np.random.randint(0, len(dataset))]
    axes[i].imshow(org_tensor.permute(1, 2, 0))
    axes[i].set_title("Original")
    axes[i].axis('off')
for i in range(5, 10):
    _, second_tensor, label = dataset[np.random.randint(0, len(dataset))]
    axes[i].imshow(second_tensor.permute(1, 2, 0)) # Display the second image␣
 ↪from the pair
    axes[i].set_title("Fake")
    axes[i].axis('off')
plt.tight_layout()
plt.show()
```

4.6 Same Person Original and Fake

[6]:
```python
# prompt: show same person signatures upto 10


# Show 10 genuine signatures for the same person

fig, axes = plt.subplots(2, 5, figsize=(15, 6))

axes = axes.flatten()


# Select a random person ID

person_id = np.random.randint(1, 56)


print(f"Showing signatures for person ID: {person_id}")


# List to store the samples we have displayed for this person

displayed_samples = []


for i in range(10):

    sample = np.random.randint(1, 24)

    # Ensure we don't display the same sample multiple times in this grid if␣
↪possible

    while sample in displayed_samples and len(displayed_samples) < 23: # Max␣
↪samples per person is 23

        sample = np.random.randint(1, 24)

    displayed_samples.append(sample)


    # Construct the image name for a genuine signature

    img_name = f'original_{person_id}_{sample}.png'

    img_path = os.path.join(original_path, img_name)
    try:
        img = Image.open(img_path).convert('RGB')
        aug = transforms.Compose([
            transforms.Resize((155, 220)),
            transforms.ToTensor()
        ])
        img_tensor = aug(img)
        axes[i].imshow(img_tensor.permute(1, 2, 0))
        axes[i].set_title(f"Sample {sample}")
        axes[i].axis('off')
    except FileNotFoundError:
        print(f"File not found: {img_path}. Skipping this sample.")
        # If file not found, leave the subplot empty or show a placeholder
        axes[i].text(0.5, 0.5, "Image not found", horizontalalignment='center',␣
 ↪verticalalignment='center')
        axes[i].axis('off')



plt.tight_layout()
plt.show()
```

```
Showing signatures for person ID: 48
```



4.7 Training Model

[8]:
```
import torch
import torch.optim as optim
from tqdm import tqdm
from torch.utils.data import DataLoader # Import DataLoader here
import os # Import the os module

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Paths
original_dir = os.path.join(sohonjit_signature_dataset_path, 'signatures/
↪full_org')
forge_dir = os.path.join(sohonjit_signature_dataset_path, 'signatures/
↪full_forg')

# Dataset & Dataloader
dataset = SigDataset(original_dir, forge_dir)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

# Model
model = SigNet().to(device)
criterion = ContrastiveLoss(margin=1.0)
optimizer = optim.Adam(model.parameters(), lr=1e-4)

# Training
```

```
num_epochs = 10

for epoch in range(num_epochs):

    loop = tqdm(dataloader, desc=f"Epoch {epoch+1}/{num_epochs}")

    for img1, img2, label in loop:

        img1, img2, label = img1.to(device), img2.to(device), label.to(device)

        optimizer.zero_grad()

        out1, out2 = model(img1, img2)

        loss = criterion(out1, out2, label)

        loss.backward()

        optimizer.step()

        loop.set_postfix(loss=loss.item())


# Save model

torch.save(model.state_dict(), "sig_net.pth")

print("Model saved!")
```

```
Epoch 1/10: 100%|| 157/157 [02:38<00:00,  1.01s/it, loss=0.234]

Epoch 2/10: 100%|| 157/157 [02:23<00:00,  1.09it/s, loss=0.0285]

Epoch 3/10: 100%|| 157/157 [02:22<00:00,  1.10it/s, loss=0.161]

Epoch 4/10: 100%|| 157/157 [02:22<00:00,  1.10it/s, loss=0.000825]

Epoch 5/10: 100%|| 157/157 [02:26<00:00,  1.07it/s, loss=0.000256]

Epoch 6/10: 100%|| 157/157 [02:21<00:00,  1.11it/s, loss=0.0366]

Epoch 7/10: 100%|| 157/157 [02:21<00:00,  1.11it/s, loss=0.367]

Epoch 8/10: 100%|| 157/157 [02:22<00:00,  1.11it/s, loss=0.000271]

Epoch 9/10: 100%|| 157/157 [02:21<00:00,  1.11it/s, loss=9.89e-5]

Epoch 10/10: 100%|| 157/157 [02:21<00:00,  1.11it/s, loss=8.5e-5]


Model saved!
```

4.8 Inference

[23]:
```
import torch.nn.functional as F
from torchvision import transforms
from PIL import Image, ImageOps # Import ImageOps explicitly
import numpy as np
def predict_genuine_or_forged(model, original_images, test_image,␣
 ↪device, threshold=1.2):

    transform = transforms.Compose([
        transforms.Lambda(lambda img: ImageOps.invert(img)),
        transforms.Resize((155, 220)),
        transforms.ToTensor()
    ])
    test_tensor = transform(test_image).unsqueeze(0).to(device)
    all_distances = []
    valid_distances = []
    for idx, img in enumerate(original_images):
        img_tensor = transform(img).unsqueeze(0).to(device)
        with torch.no_grad():
            out1, out2 = model(img_tensor, test_tensor)
            dist = F.pairwise_distance(out1, out2).item()
            print(f"Distance to Reference {idx+1}: {dist:.4f}")
            all_distances.append(dist)
            if dist < threshold:
                valid_distances.append(dist)
    # Use average of at least 3 valid distances
    if len(valid_distances) >= 3:
        avg_distance = sum(valid_distances[:3]) / 3  # Take first 3␣
 ↪valid matches
        prediction = "Genuine"
    else:
        # Not enough good matches
        avg_distance = float('inf') if not valid_distances else␣
 ↪min(valid_distances)
        prediction = "Forged"
    return prediction, avg_distance, all_distances
```

4.9 Taking images From user

[15]:
```
import io
from PIL import Image
# Upload 5 known signatures of a person
print("Upload 5 known genuine signatures of the same person:")
# Call files.upload() once and store the result
uploaded_files = files.upload()
known_signatures = [Image.open(io.BytesIO(file_content)) for␣
 ↪file_content in uploaded_files.values()]
# Upload 1 test signature
print("\nUpload 1 test signature to check:")
test_sig = list(files.upload().values())[0]
test_image = Image.open(io.BytesIO(test_sig))
```

```
Upload 5 known genuine signatures of the same person:
```

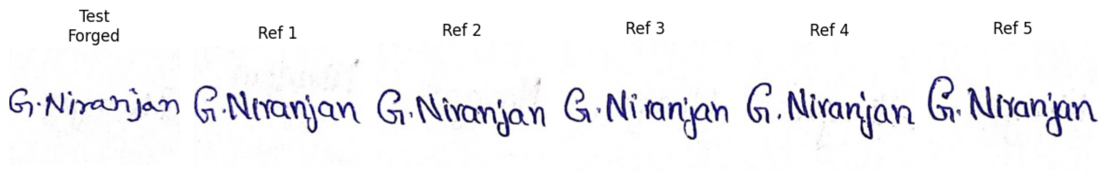
```
<IPython.core.display.HTML object>

Saving IMG_20250513_203643.jpg to IMG_20250513_203643 (1).jpg
Saving IMG_20250513_203720.jpg to IMG_20250513_203720 (1).jpg
Saving IMG_20250513_203758.jpg to IMG_20250513_203758 (1).jpg
Saving IMG_20250513_203840.jpg to IMG_20250513_203840 (1).jpg
Saving IMG_20250513_203901.jpg to IMG_20250513_203901 (1).jpg


Upload 1 test signature to check:

<IPython.core.display.HTML object>

Saving IMG_20250513_204013.jpg to IMG_20250513_204013 (1).jpg
```

[28]:
```
"Average Distance (from 3 best): {score:.4f}")
print(f"All Distances: {[f'{d:.4f}' for d in all_distances]}")
print("--------------------------------------------------------")
```

```
Distance to Reference 1: 0.2810
Distance to Reference 2: 0.4319
Distance to Reference 3: 0.2522
Distance to Reference 4: 0.2150
Distance to Reference 5: 0.2242


--- Inference Using 3 Valid Matches (Threshold: 0.13) ---
Prediction: Forged
Average Distance (from 3 best): 0.2150
All Distances: ['0.2810', '0.4319', '0.2522', '0.2150', '0.2242']
--------------------------------------------------------
```

4.10 Visualizing The User Images

[29]:
```python
import matplotlib.pyplot as plt


plt.figure(figsize=(12, 6))

plt.subplot(1, 6, 1)

plt.imshow(test_image)

plt.title(f"Test\n{prediction}")

plt.axis("off")


for i, sig in enumerate(known_signatures):

    plt.subplot(1, 6, i+2)

    plt.imshow(sig)

    plt.title(f"Ref {i+1}")

    plt.axis("off")


plt.tight_layout()

plt.show()
```



4.11 Saving the Model

[34]:
```
torch.save(model.state_dict(), "sig_net.pth")  # Save model state␣
 ↪dictionary
print("Model saved!")
```

```
Model saved!
```

4.12 Uploading Model to drive

[39]:
```
from google.colab import drive
import os

# Mount your Google Drive
drive.mount('/content/drive')

# Create the directory if it doesn't exist
model_save_path = '/content/drive/My Drive/Colab Notebooks/Models/
 ↪sig_net.pth'
os.makedirs(os.path.dirname(model_save_path), exist_ok=True)

# Now save the model
torch.save(model.state_dict(), model_save_path)
print("Model saved!")
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount,␣
 ↪call
drive.mount("/content/drive", force_remount=True).
Model saved!
```

Chapter 5

Conclusion and Future Scope

Conclusion

This project presents a robust solution for offline signature verification, tackling the challenge of distinguishing genuine signatures from forgeries in critical fields like finance, law, and healthcare. By leveraging Siamese networks with a triplet loss function, the system effectively learns to compare signature pairs, ensuring genuine signatures are recognized as similar while forgeries are flagged as distinct. The integration of SigScatNet enhances this by extracting invariant, fine-grained features, making the model resilient to handwriting variations and skilled forgeries. Experimental results show it's a reliable tool for fraud detection, identity verification, and document authentication, offering a practical solution for real-world applications.

Future Work

Real-Time Verification: Enable instant fraud detection during transactions, especially useful in banking and online payments.

Multi-Modal Integration: Combine signature verification with other biometric systems such as fingerprints or facial recognition for enhanced security.

Mobile Application Development: Build an intuitive mobile app for on-the-go verification, allowing users to authenticate signatures directly from smartphones.

Cloud-Based Platform: Develop a scalable cloud-based service to provide secure, remote access to verification tools across industries.