

peepu_urop

November 29, 2025

```
[8]: # === ResNet-50 TRAINING (WINDOWS + JUPYTER) - FINAL CODE ===

import os
import random
import time
import cv2
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torchvision import models, transforms
from tqdm import tqdm
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
from torch.optim.lr_scheduler import ReduceLROnPlateau
```

```
[9]: # -----
# REPRODUCIBILITY
# -----
seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(seed)
```

```
[10]: # -----
# CONFIG
# -----
DATASET_DIR = r"E:\Shared_folder\Peepu_UROP\CASIA-dataset" # <-- update to
↳ your dataset folder
attack_ratio = 0.5
BATCH_SIZE = 16
EPOCHS = 25
LR = 1e-4
WEIGHT_DECAY = 1e-4
```

```

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
CHECKPOINT_PATH = "resnet50_best.pth"
NUM_WORKERS = 0 # safe for Windows + Jupyter
EARLY_STOPPING_PATIENCE = 6
GRAD_CLIP_NORM = 3.0

print("Using device:", DEVICE)

```

Using device: cuda

```

[11]: # -----
# DATASET CLASS (FIXED & ROBUST)
# -----
class FingerprintDataset(Dataset):
    def __init__(self, samples, transform=None):
        """
        samples: list of tuples (img_path, label)
        label: 0 -> Real, 1 -> Reconstructed/Attack
        """
        self.samples = samples
        self.transform = transform

    def __len__(self):
        return len(self.samples)

    def _apply_simulated_attack(self, image):
        """Simulate realistic attack artifacts on a grayscale image (numpy 2D).
        ↪ """
        # blur
        blur_val = int(np.random.choice([3, 5, 7]))
        sigma = float(np.random.uniform(0.3, 2.5))
        attacked = cv2.GaussianBlur(image, (blur_val, blur_val), sigmaX=sigma)

        # contrast/brightness
        alpha = float(np.random.uniform(0.6, 1.3))
        beta = int(np.random.randint(-25, 25))
        attacked = attacked.astype(np.float32) * alpha + beta

        # noise sometimes
        if np.random.rand() > 0.35:
            noise_sigma = float(np.random.uniform(3.0, 18.0))
            noise = np.random.normal(0, noise_sigma, attacked.shape).astype(np.
            ↪float32)
            attacked = attacked + noise

        # occasional JPEG artifacts
        if np.random.rand() > 0.6:

```

```

        encode_param = [int(cv2.IMWRITE_JPEG_QUALITY), int(np.random.
↪uniform(30, 85))]
        _, encimg = cv2.imencode('.jpg', np.clip(attacked, 0, 255).
↪astype(np.uint8), encode_param)
        attacked = cv2.imdecode(encimg, cv2.IMREAD_GRAYSCALE).astype(np.
↪float32)

        attacked = np.clip(attacked, 0, 255).astype(np.uint8)
        return attacked

    def __getitem__(self, idx):
        img_path, label = self.samples[idx]
        try:
            image = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
            if image is None:
                raise IOError("Failed to load image (cv2.imread returned None)")

            # resize to model input size
            image = cv2.resize(image, (224, 224))

            # if marked as attack, simulate attack
            if label == 1:
                image = self._apply_simulated_attack(image)

            # convert grayscale->RGB (3 channels) as models expect 3-channel
↪input
            image = cv2.cvtColor(image, cv2.COLOR_GRAY2RGB)

            if self.transform is not None:
                image = self.transform(image)

            return image, torch.tensor(label, dtype=torch.long)

        except Exception as e:
            # fallback dummy item (keeps DataLoader stable)
            print(f"[Warning] Error loading {img_path}: {e}")
            return torch.randn(3, 224, 224), torch.tensor(0, dtype=torch.long)

```

```

[12]: # -----
# TRANSFORMS
# -----
train_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(15),
    transforms.RandomPerspective(distortion_scale=0.25, p=0.5),

```

```

        transforms.ColorJitter(brightness=0.25, contrast=0.25, saturation=0.15),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                               [0.229, 0.224, 0.225])
    ])

test_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                          [0.229, 0.224, 0.225])
])

```

```

[13]: # -----
# LOAD IMAGE PATHS
# -----
all_real_images = []

if not os.path.isdir(DATASET_DIR):
    raise FileNotFoundError(f"DATASET_DIR not found: {DATASET_DIR}")

for subject in sorted(os.listdir(DATASET_DIR)):
    subj_path = os.path.join(DATASET_DIR, subject)
    if not os.path.isdir(subj_path):
        continue

    for hand in ['L', 'R']:
        hand_path = os.path.join(subj_path, hand)
        if not os.path.isdir(hand_path):
            continue

        for img_name in os.listdir(hand_path):
            if img_name.lower().endswith(('.bmp', '.png', '.jpg', '.jpeg', '.
↳tif', '.tiff')):
                all_real_images.append(os.path.join(hand_path, img_name))

print(f"Found {len(all_real_images)} real images.")

if len(all_real_images) < 10:
    raise RuntimeError("Dataset looks too small; please check DATASET_DIR and_
↳folder structure.")

# deterministic-ish shuffle
random.shuffle(all_real_images)
split_idx = int(0.8 * len(all_real_images))
train_real_paths = all_real_images[:split_idx]

```

```

test_real_paths = all_real_images[split_idx:]

# create samples: (path, label)
train_samples = [(p, 0) for p in train_real_paths]
test_samples = [(p, 0) for p in test_real_paths]

num_train_attacks = int(len(train_real_paths) * attack_ratio)
num_test_attacks = int(len(test_real_paths) * attack_ratio)

# sample without replacement where possible
if num_train_attacks <= len(train_real_paths):
    attack_sources_train = random.sample(train_real_paths, num_train_attacks)
else:
    attack_sources_train = [random.choice(train_real_paths) for _ in
↪range(num_train_attacks)]

if num_test_attacks <= len(test_real_paths):
    attack_sources_test = random.sample(test_real_paths, num_test_attacks)
else:
    attack_sources_test = [random.choice(test_real_paths) for _ in
↪range(num_test_attacks)]

train_samples += [(p, 1) for p in attack_sources_train]
test_samples += [(p, 1) for p in attack_sources_test]

random.shuffle(train_samples)
random.shuffle(test_samples)

train_dataset = FingerprintDataset(train_samples, transform=train_transform)
test_dataset = FingerprintDataset(test_samples, transform=test_transform)

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True,
↪num_workers=NUM_WORKERS)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False,
↪num_workers=NUM_WORKERS)

print(f"Train size: {len(train_dataset)}, Test size: {len(test_dataset)}")

```

Found 20000 real images.

Train size: 24000, Test size: 6000

Train size: 24000, Test size: 6000

```

[14]: # -----
# MODEL: ResNet-50 (fine-tuning)
# -----
try:
    # new torchvision API

```

```

weights = models.ResNet50_Weights.DEFAULT
model = models.resnet50(weights=weights)
except Exception:
    # fallback for older torchvision
    model = models.resnet50(pretrained=True)

# freeze all
for p in model.parameters():
    p.requires_grad = False

# unfreeze later layers
for p in model.layer2.parameters():
    p.requires_grad = True
for p in model.layer3.parameters():
    p.requires_grad = True
for p in model.layer4.parameters():
    p.requires_grad = True

# replace classifier head
num_fts = model.fc.in_features
model.fc = nn.Sequential(
    nn.Dropout(0.5),
    nn.Linear(num_fts, 512),
    nn.ReLU(),
    nn.Dropout(0.4),
    nn.Linear(512, 2)
)

model = model.to(DEVICE)

```

```

[15]: # -----
# LOSS / OPTIMIZER / SCHEDULER
# -----
# label_smoothing exists in newer PyTorch; use try/except for compatibility
try:
    criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
except TypeError:
    criterion = nn.CrossEntropyLoss()

params_to_optimize = [p for p in model.parameters() if p.requires_grad]
optimizer = optim.Adam(params_to_optimize, lr=LR, weight_decay=WEIGHT_DECAY)

# ReduceLROnPlateau without verbose (compatible across versions)
scheduler = ReduceLROnPlateau(optimizer, mode="min", patience=2, factor=0.3)

```

```

[16]: # -----
# TRAINING LOOP (with early stopping + gradient clipping)

```

```

# -----
best_val_loss = float("inf")
train_losses, val_losses = [], []
train_accs, val_accs = [], []

no_improve_epochs = 0
start_time = time.time()

for epoch in range(EPOCHS):
    epoch_start = time.time()
    print(f"\n===== Epoch {epoch+1}/{EPOCHS} =====")
    model.train()
    running_loss = 0.0
    running_correct = 0
    running_total = 0

    for imgs, labels in tqdm(train_loader, desc="Training", leave=False):
        imgs, labels = imgs.to(DEVICE), labels.to(DEVICE)

        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        loss.backward()

        torch.nn.utils.clip_grad_norm_(params_to_optimize, GRAD_CLIP_NORM)
        optimizer.step()

        running_loss += loss.item() * imgs.size(0)
        _, preds = torch.max(outputs, 1)
        running_correct += (preds == labels).sum().item()
        running_total += labels.size(0)

    if running_total == 0:
        print("[Warning] No training samples processed this epoch.")
        continue

    train_loss = running_loss / running_total
    train_acc = running_correct / running_total
    train_losses.append(train_loss)
    train_accs.append(train_acc)

    # Validation
    model.eval()
    val_loss = 0.0
    val_correct = 0
    val_total = 0

```

```

with torch.no_grad():
    for imgs, labels in test_loader:
        imgs, labels = imgs.to(DEVICE), labels.to(DEVICE)
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        val_loss += loss.item() * imgs.size(0)
        _, preds = torch.max(outputs, 1)
        val_correct += (preds == labels).sum().item()
        val_total += labels.size(0)

if val_total == 0:
    print("[Warning] No validation samples available.")
    break

val_loss = val_loss / val_total
val_acc = val_correct / val_total
val_losses.append(val_loss)
val_accs.append(val_acc)

print(f"Train Loss: {train_loss:.4f} | Train Acc: {train_acc:.4f}")
print(f"Val Loss: {val_loss:.4f} | Val Acc: {val_acc:.4f}")

# scheduler step with manual verbose emulation
old_lr = optimizer.param_groups[0]['lr']
scheduler.step(val_loss)
new_lr = optimizer.param_groups[0]['lr']
if new_lr != old_lr:
    print(f"LR reduced: {old_lr:.6f} -> {new_lr:.6f}")

# checkpoint & early stopping
if val_loss < best_val_loss - 1e-4:
    best_val_loss = val_loss
    torch.save(model.state_dict(), CHECKPOINT_PATH)
    print(" Saved best checkpoint")
    no_improve_epochs = 0
else:
    no_improve_epochs += 1
    print(f"No improvement for {no_improve_epochs} epoch(s).")

if no_improve_epochs >= EARLY_STOPPING_PATIENCE:
    print("Early stopping: no improvement for", EARLY_STOPPING_PATIENCE,
↪ "epochs.")
    break

epoch_time = time.time() - epoch_start
print(f"Epoch time: {epoch_time:.1f}s")

```

```
total_time = time.time() - start_time
print(f"\nTraining finished in {total_time/60:.2f} minutes.")
```

===== Epoch 1/25 =====

Train Loss: 0.2572 | Train Acc: 0.9702
Val Loss: 0.2162 | Val Acc: 0.9912
Saved best checkpoint
Epoch time: 412.3s

===== Epoch 2/25 =====

Train Loss: 0.2315 | Train Acc: 0.9863
Val Loss: 0.2115 | Val Acc: 0.9945
Saved best checkpoint
Epoch time: 246.9s

===== Epoch 3/25 =====

Train Loss: 0.2298 | Train Acc: 0.9866
Val Loss: 0.2116 | Val Acc: 0.9955
No improvement for 1 epoch(s).
Epoch time: 246.1s

===== Epoch 4/25 =====

Train Loss: 0.2251 | Train Acc: 0.9881
Val Loss: 0.2133 | Val Acc: 0.9943
No improvement for 2 epoch(s).
Epoch time: 245.9s

===== Epoch 5/25 =====

Train Loss: 0.2240 | Train Acc: 0.9890
Val Loss: 0.2123 | Val Acc: 0.9947
LR reduced: 0.000100 -> 0.000030
No improvement for 3 epoch(s).
Epoch time: 246.6s

===== Epoch 6/25 =====

Train Loss: 0.2172 | Train Acc: 0.9924
Val Loss: 0.2089 | Val Acc: 0.9957
Saved best checkpoint
Epoch time: 247.1s

==== Epoch 7/25 ====

Train Loss: 0.2191 | Train Acc: 0.9909
Val Loss: 0.2113 | Val Acc: 0.9950
No improvement for 1 epoch(s).
Epoch time: 247.0s

==== Epoch 8/25 ====

Train Loss: 0.2167 | Train Acc: 0.9922
Val Loss: 0.2087 | Val Acc: 0.9950
Saved best checkpoint
Epoch time: 246.4s

==== Epoch 9/25 ====

Train Loss: 0.2148 | Train Acc: 0.9933
Val Loss: 0.2076 | Val Acc: 0.9965
Saved best checkpoint
Epoch time: 248.1s

==== Epoch 10/25 ====

Train Loss: 0.2174 | Train Acc: 0.9921
Val Loss: 0.2093 | Val Acc: 0.9952
No improvement for 1 epoch(s).
Epoch time: 246.6s

==== Epoch 11/25 ====

Train Loss: 0.2164 | Train Acc: 0.9926
Val Loss: 0.2053 | Val Acc: 0.9972
Saved best checkpoint
Epoch time: 247.3s

==== Epoch 12/25 ====

Train Loss: 0.2162 | Train Acc: 0.9925
Val Loss: 0.2076 | Val Acc: 0.9963
No improvement for 1 epoch(s).
Epoch time: 247.1s

==== Epoch 13/25 ====

Train Loss: 0.2149 | Train Acc: 0.9928
Val Loss: 0.2085 | Val Acc: 0.9958
No improvement for 2 epoch(s).
Epoch time: 246.7s

==== Epoch 14/25 ====

Train Loss: 0.2170 | Train Acc: 0.9923
Val Loss: 0.2099 | Val Acc: 0.9952
LR reduced: 0.000030 -> 0.000009
No improvement for 3 epoch(s).
Epoch time: 246.6s

==== Epoch 15/25 ====

Train Loss: 0.2131 | Train Acc: 0.9936
Val Loss: 0.2067 | Val Acc: 0.9967
No improvement for 4 epoch(s).
Epoch time: 246.5s

==== Epoch 16/25 ====

Train Loss: 0.2117 | Train Acc: 0.9946
Val Loss: 0.2053 | Val Acc: 0.9967
No improvement for 5 epoch(s).
Epoch time: 246.4s

==== Epoch 17/25 ====

Train Loss: 0.2111 | Train Acc: 0.9949
Val Loss: 0.2058 | Val Acc: 0.9967
LR reduced: 0.000009 -> 0.000003
No improvement for 6 epoch(s).
Early stopping: no improvement for 6 epochs.

Training finished in 72.67 minutes.

```

[17]: # -----
# EVALUATION
# -----
# load best model (if saved)
if os.path.exists(CHECKPOINT_PATH):
    model.load_state_dict(torch.load(CHECKPOINT_PATH, map_location=DEVICE))
model.eval()

y_true, y_pred = [], []

with torch.no_grad():
    for imgs, labels in tqdm(test_loader, desc="Evaluating"):
        imgs = imgs.to(DEVICE)
        outputs = model(imgs)
        _, preds = torch.max(outputs, 1)

        # ensure CPU numpy arrays
        y_true.extend(labels.cpu().numpy())
        y_pred.extend(preds.cpu().numpy())

print("\nClassification Report:")
print(classification_report(y_true, y_pred, target_names=["Real",
↪ "Reconstructed"]))

cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(5,4))
sns.heatmap(cm, annot=True, cmap='Blues', fmt='d',
            xticklabels=["Real", "Recon"], yticklabels=["Real", "Recon"])
plt.title("Confusion Matrix")
plt.show()

# Plot training curves
plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.plot(train_losses, label='train_loss')
plt.plot(val_losses, label='val_loss')
plt.title('Loss')
plt.legend()
plt.subplot(1,2,2)
plt.plot(train_accs, label='train_acc')
plt.plot(val_accs, label='val_acc')
plt.title('Accuracy')
plt.legend()
plt.tight_layout()
plt.show()

```

Evaluating: 100%| | 375/375 [00:21<00:00, 17.73it/s]

Classification Report:

| | precision | recall | f1-score | support |
|---------------|-----------|--------|----------|---------|
| Real | 0.99 | 1.00 | 1.00 | 4000 |
| Reconstructed | 1.00 | 0.98 | 0.99 | 2000 |
| accuracy | | | 0.99 | 6000 |
| macro avg | 1.00 | 0.99 | 0.99 | 6000 |
| weighted avg | 0.99 | 0.99 | 0.99 | 6000 |



