```
!pip install -U transformers
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.
Requirement already satisfied: filelock in /usr/local/lib/python3.12/d
Requirement already satisfied: huggingface-hub<1.0,>=0.34.0 in /usr/lo
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.1
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/pytho
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.1
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/pyt
Requirement already satisfied: requests in /usr/local/lib/python3.12/d
Requirement already satisfied: tokenizers<=0.23.0,>=0.22.0 in /usr/loc
Requirement already satisfied: safetensors>=0.4.3 in /usr/local/lib/py
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.12
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/pyth
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/loca
Requirement already satisfied: hf-xet<2.0.0,>=1.1.3 in /usr/local/lib/
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/py
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/py
```

Local Inference on GPU

Model page: https://huggingface.co/google/paligemma-3b-mix-224

⚠️ If the generated code snippets do not work, please open an issue on either the model repo and/or on huggingface.js 🙏

The model you are trying to use is gated. Please make sure you have access to it by visiting the model page.To run inference, either set HF_TOKEN in your environment variables/ Secrets or run the following cell to login. 🤗

```
from huggingface_hub import login
login(new_session=False)
```

```
# Use a pipeline as a high-level helper
from transformers import pipeline

pipe = pipeline("image-text-to-text", model="google/paligemma-3b-mix-
```

```
config.json: 100%                                   1.03k/1.03k [00:00<00:00, 125kB/s]
model.safetensors.index.json: 100%                  62.6k/62.6k [00:00<00:00, 7.37MB/s]
Fetching 3 files: 100%                              3/3 [00:32<00:00, 13.59s/it]
model-00003-of-00003.safetensors: 100%              1.74G/1.74G [00:15<00:00, 67.6MB/s]
model-00001-of-00003.safetensors: 100%              4.95G/4.95G [00:30<00:00, 219MB/s]
model-00002-of-00003.safetensors: 100%              5.00G/5.00G [00:31<00:00, 278MB/s]
Loading checkpoint shards: 100%                     3/3 [00:02<00:00,  1.12s/it]
generation_config.json: 100%                        137/137 [00:00<00:00, 17.8kB/s]
preprocessor_config.json: 100%                      699/699 [00:00<00:00, 98.9kB/s]
Using a slow image processor as `use_fast` is unset and a slow processo
tokenizer_config.json: 100%                         40.0k/40.0k [00:00<00:00, 5.05MB/s]
tokenizer.model: 100%                               4.26M/4.26M [00:00<00:00, 8.52MB/s]
tokenizer.json: 100%                                17.5M/17.5M [00:00<00:00, 350kB/s]
added_tokens.json: 100%                             24.0/24.0 [00:00<00:00, 3.52kB/s]
```

```
# Load model directly
from transformers import AutoProcessor, AutoModelForVision2Seq

processor = AutoProcessor.from_pretrained("google/paligemma-3b-mix-22
model = AutoModelForVision2Seq.from_pretrained("google/paligemma-3b-m
```

```
/usr/local/lib/python3.12/dist-packages/transformers/models/auto/modeli
  warnings.warn(
```

Loading checkpoint shards: 100% 3/3 [00:00<00:00, 33.29it/s]

```
import os, gc, re, json, random, warnings
import numpy as np
import pandas as pd
from tqdm.auto import tqdm
from PIL import Image

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from transformers import AutoProcessor, PaliGemmaForConditionalGenerati

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score, precision_score,

warnings.filterwarnings('ignore')
print("✅ Imports complete")

# ============================================================================
# CELL 3: SETUP
# ============================================================================
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed_all(SEED)

from google.colab import drive
drive.mount('/content/drive')

print(f"GPU: {torch.cuda.get_device_name(0)}")
print(f"CUDA: {torch.cuda.is_available()}")
print("✅ Setup complete")
```

```
# ======================================================================
# CELL 4: CONFIG
# ======================================================================
class Config:
    VIA_IARC_ROOT = "/content/drive/MyDrive/IARCImageBankVIA"
    VIA_IARC_META = "/content/drive/MyDrive/IARCImageBankVIA/meta_data.
    VIA_JHPIEGO_ROOT = "/content/drive/MyDrive/via_cropped+index"
    OUTPUT_DIR = "/content/drive/MyDrive/VIA_PaliGemma_Fixed"
    CHECKPOINT_DIR = os.path.join(OUTPUT_DIR, "checkpoints")
    METRICS_DIR = os.path.join(OUTPUT_DIR, "metrics")

    MODEL_NAME = "google/paligemma-3b-mix-224"
    IMG_SIZE = 224
    BATCH_SIZE = 4
    EPOCHS = 10
    LEARNING_RATE = 1e-3
    WEIGHT_DECAY = 0.01
    TEST_SIZE = 0.2
    VAL_SIZE = 0.1
    POSITIVE_CLASSES = ['Positive', 'Suspicious']
    DEVICE = torch.device("cuda")

cfg = Config()
os.makedirs(cfg.CHECKPOINT_DIR, exist_ok=True)
os.makedirs(cfg.METRICS_DIR, exist_ok=True)
print(f"✅ Config set")

# ======================================================================
# CELL 5: LOAD DATA
# ======================================================================
def load_via_datasets():
    JHPIEGO_LABELS = {
        'Negative': [3,4,6,9,12,14,15,16,19,22,26,27,28,30,31,33,34,37,
        'Positive': [1,2,8,10,11,13,17,18,20,21,23,24,29,32,35,36,38,39
        'Suspicious': [5,7,25,55,65,89]
    }
    card_to_label = {card: label for label, cards in JHPIEGO_LABELS.ite

    jhpiego_records = []
    if os.path.exists(cfg.VIA_JHPIEGO_ROOT):
        for filename in os.listdir(cfg.VIA_JHPIEGO_ROOT):
            if not filename.startswith("via_img_"): continue
            match = re.search(r'via_img_(\d+)', filename)
            if match:
                card_num = int(match.group(1))
```

```
                card_num = int(match.group(1))
                if card_num in card_to_label:
                    jhpiego_records.append({
                        "CaseNumber": f"JHPIEGO_{card_num}",
                        "image_path": os.path.join(cfg.VIA_JHPIEGO_ROOT
                        "VIA_simple": card_to_label[card_num],
                        "Source": "JHPIEGO"
                    })
        jhpiego_data = pd.DataFrame(jhpiego_records)
        print(f"JHPIEGO: {len(jhpiego_data)}")
    else:
        jhpiego_data = pd.DataFrame()

    iarc_data = pd.DataFrame()
    if os.path.exists(cfg.VIA_IARC_META):
        iarc_meta = pd.read_excel(cfg.VIA_IARC_META)
        iarc_meta["CaseNumber"] = iarc_meta["CaseNumber"].astype(int)

        def simplify_via(x):
            s = str(x).strip().lower()
            if "suspicious" in s: return "Suspicious"
            elif "positive" in s: return "Positive"
            else: return "Negative"

        iarc_meta["VIA_simple"] = iarc_meta["VIA"].apply(simplify_via)

        iarc_records = []
        for entry in os.listdir(cfg.VIA_IARC_ROOT):
            case_dir = os.path.join(cfg.VIA_IARC_ROOT, entry)
            if not os.path.isdir(case_dir): continue
            digits = re.findall(r"\d+", entry)
            if not digits: continue
            case_num = int(digits[0])

            for f in os.listdir(case_dir):
                if f.lower().endswith(('.jpg', '.jpeg', '.png')):
                    if f.replace('.jpg', '').replace('.jpeg', '').repla
                        iarc_records.append({
                            "CaseNumber": case_num,
                            "image_path": os.path.join(case_dir, f),
                            "Source": "IARC"
                        })
                        break

        iarc_images = pd.DataFrame(iarc_records)
        iarc_data = pd.merge(iarc_meta, iarc_images, on="CaseNumber", h
```

```
            iarc_data = pd.merge(iarc_meta, iarc_images, on="CaseNumber", h
            iarc_data["CaseNumber"] = iarc_data["CaseNumber"].astype(str)
            print(f"IARC: {len(iarc_data)}")

        combined = pd.concat([iarc_data, jhpiego_data], ignore_index=True)
        combined['binary_label'] = combined['VIA_simple'].apply(lambda x: '
        combined['label'] = (combined['binary_label'] == 'Abnormal').astype
        valid_mask = combined['image_path'].apply(os.path.isfile)
        combined = combined[valid_mask].reset_index(drop=True)
        return combined

    data_df = load_via_datasets()
    print(f"\nTotal: {len(data_df)} | Normal: {(data_df['label']==0).sum()}

    train_val_df, test_df = train_test_split(data_df, test_size=cfg.TEST_SI
    train_df, val_df = train_test_split(train_val_df, test_size=cfg.VAL_SIZ
    print(f"Train: {len(train_df)} | Val: {len(val_df)} | Test: {len(test_d
    print("✅ Data loaded")

    # ============================================================================
    # CELL 6: DATASET
    # ============================================================================
    class VIADataset(Dataset):
        def __init__(self, dataframe, processor, img_size=224):
            self.df = dataframe.reset_index(drop=True)
            self.processor = processor
            self.img_size = img_size

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

        def __getitem__(self, idx):
            row = self.df.iloc[idx]
            try:
                image = Image.open(row['image_path']).convert('RGB')
                if image.size != (self.img_size, self.img_size):
                    image = image.resize((self.img_size, self.img_size), Im
            except:
                image = Image.new('RGB', (self.img_size, self.img_size), (1

            inputs = self.processor(images=image, return_tensors="pt")
            return {
                'pixel_values': inputs['pixel_values'].squeeze(0),
                'label': torch.tensor(int(row['label']), dtype=torch.long)
            }
```

```
print("✅ Dataset class created")

# ============================================================================
# CELL 7: LOAD PALIGEMMA
# ============================================================================
torch.cuda.empty_cache()
gc.collect()

print("Loading PaliGemma...")
processor = AutoProcessor.from_pretrained(cfg.MODEL_NAME)
print("  ✓ Processor loaded")

paligemma_model = PaliGemmaForConditionalGeneration.from_pretrained(
    cfg.MODEL_NAME,
    torch_dtype=torch.float16,
    device_map="auto"
)
print("  ✓ Model loaded")
print(f"GPU Memory: {torch.cuda.memory_allocated()/1e9:.2f} GB")
print("✅ PaliGemma ready")

# ============================================================================
# CELL 8: CLASSIFIER (FIXED)
# ============================================================================
class PaliGemmaClassifier(nn.Module):
    """Fixed classifier with proper tensor handling"""

    def __init__(self, base_model, num_classes=2):
        super().__init__()
        self.vision_model = base_model.vision_tower

        # Freeze vision encoder
        for param in self.vision_model.parameters():
            param.requires_grad = False

        print("Detecting architecture...")

        # Test forward pass to understand output structure
        with torch.no_grad():
            # Ensure dummy input matches model's dtype (float16)
            # Removed .to(cfg.DEVICE) as device_map='auto' should handl
            dummy = torch.randn(1, 3, 224, 224, dtype=torch.float16)
            out = self.vision_model(dummy)

            # Extract features
```

```
            if hasattr(out, 'last_hidden_state'):
                features = out.last_hidden_state
            elif hasattr(out, 'pooler_output'):
                features = out.pooler_output
            elif isinstance(out, tuple):
                features = out[0]
            else:
                features = out

            # Ensure features are float32 for classifier
            features = features.float()

            print(f"  Vision output shape: {features.shape}")
            print(f"  Vision output dtype: {features.dtype}")

            # Determine dimensions
            if len(features.shape) == 3:
                # [batch, sequence, features]
                batch_size, seq_len, feature_dim = features.shape
                self.feature_dim = feature_dim
                self.use_pooling = True
                print(f"  Has sequence: True (seq_len={seq_len})")
            elif len(features.shape) == 2:
                # [batch, features]
                batch_size, feature_dim = features.shape
                self.feature_dim = feature_dim
                self.use_pooling = False
                print(f"  Has sequence: False")
            else:
                raise ValueError(f"Unexpected feature shape: {features.

        print(f"  Feature dim: {self.feature_dim}")

        # Build classifier
        if self.use_pooling:
            # Need to pool sequence dimension
            self.pool = nn.AdaptiveAvgPool1d(1)
            self.classifier = nn.Sequential(
                nn.Flatten(),
                nn.Linear(self.feature_dim, 512),
                nn.ReLU(),
                nn.Dropout(0.3),
                nn.Linear(512, 256),
                nn.ReLU(),
                nn.Dropout(0.2),
```

```
                nn.Linear(256, num_classes)
            )
        else:
            # Already pooled
            self.classifier = nn.Sequential(
                nn.Linear(self.feature_dim, 512),
                nn.ReLU(),
                nn.Dropout(0.3),
                nn.Linear(512, 256),
                nn.ReLU(),
                nn.Dropout(0.2),
                nn.Linear(256, num_classes)
            )

    def forward(self, pixel_values):
        """Forward pass with proper error handling"""

        # Ensure input is on the correct device
        pixel_values = pixel_values.to(cfg.DEVICE)

        # Convert input to model's expected dtype (float16) before pass
        if pixel_values.dtype != torch.float16:
            pixel_values = pixel_values.to(torch.float16)

        # Extract features (frozen)
        with torch.no_grad():
            out = self.vision_model(pixel_values)

            if hasattr(out, 'last_hidden_state'):
                features = out.last_hidden_state
            elif hasattr(out, 'pooler_output'):
                features = out.pooler_output
            elif isinstance(out, tuple):
                features = out[0]
            else:
                features = out

        # Ensure features are in float32 for classifier
        features = features.float()

        # Pool if needed
        if self.use_pooling:
            # Shape: [batch, seq, features] -> [batch, features, seq]
            features = features.transpose(1, 2)
            # Pool: [batch, features, seq] -> [batch, features, 1]
```

```
                features = self.pool(features)
                # Flatten will handle the rest

            # Classify
            logits = self.classifier(features)

            return logits

    # Create classifier
    print("\nCreating classifier...")
    classifier = PaliGemmaClassifier(paligemma_model, num_classes=2).to(cfg

    trainable = sum(p.numel() for p in classifier.parameters() if p.require
    total = sum(p.numel() for p in classifier.parameters())
    print(f"\nTrainable: {trainable:,} ({100*trainable/total:.2f}%)वरुन")
    print(f"Total: {total:,}")
    print("✅ Classifier ready")

    # ============================================================================
    # CELL 9: DATA LOADERS
    # ============================================================================
    train_dataset = VIADataset(train_df, processor, cfg.IMG_SIZE)
    val_dataset = VIADataset(val_df, processor, cfg.IMG_SIZE)
    test_dataset = VIADataset(test_df, processor, cfg.IMG_SIZE)

    def collate_fn(batch):
        return {
            'pixel_values': torch.stack([item['pixel_values'] for item in b
            'labels': torch.stack([item['label'] for item in batch])
        }

    train_loader = DataLoader(train_dataset, batch_size=cfg.BATCH_SIZE, shu
    val_loader = DataLoader(val_dataset, batch_size=cfg.BATCH_SIZE, shuffle
    test_loader = DataLoader(test_dataset, batch_size=cfg.BATCH_SIZE, shuff

    print(f"Train: {len(train_loader)} batches | Val: {len(val_loader)} bat
    print("✅ Data loaders ready")

    # ============================================================================
    # CELL 10: TRAINING SETUP
    # ============================================================================
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.AdamW([p for p in classifier.parameters() if p.
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max
    print("✅ Training setup complete")
```

```
# ============================================================================
# CELL 11: TRAINING LOOP
# ============================================================================
print("="*80)
print("STARTING TRAINING")
print("="*80)

best_val_f1 = 0
patience_counter = 0

for epoch in range(cfg.EPOCHS):
    print(f"\n{'='*60}\nEpoch {epoch+1}/{cfg.EPOCHS}\n{'='*60}")

    # TRAIN
    classifier.train()
    train_loss = 0
    train_preds, train_labels = [], []

    pbar = tqdm(train_loader, desc="Training")
    for batch in pbar:
        try:
            pixel_values = batch['pixel_values'].to(cfg.DEVICE)
            labels = batch['labels'].to(cfg.DEVICE)

            optimizer.zero_grad()
            logits = classifier(pixel_values)
            loss = criterion(logits, labels)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(classifier.parameters(), 1.0
            optimizer.step()

            train_loss += loss.item()
            preds = torch.argmax(logits, dim=1).cpu().numpy()
            train_preds.extend(preds)
            train_labels.extend(labels.cpu().numpy())

            pbar.set_postfix({'loss': f'{train_loss/(pbar.n+1):.4f}'})
        except Exception as e:
            print(f"\nError in training batch: {e}")
            continue

    train_acc = accuracy_score(train_labels, train_preds)
    train_f1 = f1_score(train_labels, train_preds, average='binary')
    print(f"Train - Loss: {train_loss/len(train_loader):.4f} | Acc: {tr
```

```
        # VALIDATE
        classifier.eval()
        val_loss = 0
        val_preds, val_labels = [], []

        with torch.no_grad():
            pbar = tqdm(val_loader, desc="Validation")
            for batch in pbar:
                try:
                    pixel_values = batch['pixel_values'].to(cfg.DEVICE)
                    labels = batch['labels'].to(cfg.DEVICE)

                    logits = classifier(pixel_values)
                    loss = criterion(logits, labels)

                    val_loss += loss.item()
                    preds = torch.argmax(logits, dim=1).cpu().numpy()
                    val_preds.extend(preds)
                    val_labels.extend(labels.cpu().numpy())

                    pbar.set_postfix({'loss': f'{val_loss/(pbar.n+1):.4f}'}
                except Exception as e:
                    print(f"\nError in validation batch: {e}")
                    continue

        val_acc = accuracy_score(val_labels, val_preds)
        val_f1 = f1_score(val_labels, val_preds, average='binary')
        print(f"Val   - Loss: {val_loss/len(val_loader):.4f} | Acc: {val_ac

        # Save best
        if val_f1 > best_val_f1:
            best_val_f1 = val_f1
            patience_counter = 0
            torch.save({
                'epoch': epoch,
                'model_state_dict': classifier.state_dict(),
                'val_f1': val_f1,
                'val_acc': val_acc
            }, os.path.join(cfg.CHECKPOINT_DIR, 'best_model.pth'))
            print(f"✅ Saved (F1: {val_f1:.4f})")
        else:
            patience_counter += 1
            if patience_counter >= 5:
                print("⏹️ Early stopping")
                break
```

```
                break

        scheduler.step()
        torch.cuda.empty_cache()

    print(f"\n{'='*80}\nTRAINING COMPLETED\n{'='*80}")
    print(f"Best Val F1: {best_val_f1:.4f}")

    # ============================================================================
    # CELL 12: TEST EVALUATION
    # ============================================================================
    print(f"\n{'='*80}\nTEST EVALUATION\n{'='*80}")

    # Load best model (or last model if no improvement was saved)
    checkpoint_path_to_load = os.path.join(cfg.CHECKPOINT_DIR, 'best_model.
    # Load checkpoint onto CPU first to avoid immediate OOM
    checkpoint = torch.load(checkpoint_path_to_load, map_location='cpu')

    # Create a new classifier instance, then load state dict, then move to
    test_classifier = PaliGemmaClassifier(paligemma_model, num_classes=2)
    test_classifier.load_state_dict(checkpoint['model_state_dict'])
    test_classifier.to(cfg.DEVICE)

    test_classifier.eval()
    test_preds, test_labels, test_probs = [], [], []

    with torch.no_grad():
        for batch in tqdm(test_loader, desc="Testing"):
            pixel_values = batch['pixel_values'].to(cfg.DEVICE)
            labels = batch['labels'].to(cfg.DEVICE)

            logits = test_classifier(pixel_values)
            probs = torch.softmax(logits, dim=1)
            preds = torch.argmax(logits, dim=1)

            test_preds.extend(preds.cpu().numpy())
            test_labels.extend(labels.cpu().numpy())
            test_probs.extend(probs[:, 1].cpu().numpy())

    test_acc = accuracy_score(test_labels, test_preds)
    test_f1 = f1_score(test_labels, test_preds, average='binary')
    test_prec = precision_score(test_labels, test_preds, average='binary')
    test_rec = recall_score(test_labels, test_preds, average='binary')
    test_auc = roc_auc_score(test_labels, test_probs)

    print(f"\n{'='*60}\nTEST RESULTS\n{'='*60}")
```

```
print(f"\n{'='*60}\nTEST RESULTS\n{'='*60}")
print(f"Accuracy:  {test_acc*100:.2f}%")
print(f"Precision: {test_prec:.4f}")
print(f"Recall:    {test_rec:.4f}")
print(f"F1 Score:  {test_f1:.4f}")
print(f"AUC-ROC:   {test_auc:.4f}")
print(f"\nConfusion Matrix:\n{confusion_matrix(test_labels, test_preds)
print(f"\n{classification_report(test_labels, test_preds, target_names=
print("="*60)

results = {'test_accuracy': float(test_acc), 'test_f1': float(test_f1),
with open(os.path.join(cfg.METRICS_DIR, 'results.json'), 'w') as f:
    json.dump(results, f, indent=2)

print(f"\n✅ Results saved to {cfg.METRICS_DIR}")

# ============================================================================
# CELL 13: SAMPLE PREDICTIONS
# ============================================================================
print(f"\n{'='*80}\nSAMPLE PREDICTIONS\n{'='*80}")

indices = np.random.choice(len(test_df), min(5, len(test_df)), replace=

test_classifier.eval() # Use test_classifier for predictions
with torch.no_grad():
    for i, idx in enumerate(indices):
        row = test_df.iloc[idx]
        image = Image.open(row['image_path']).convert('RGB')
        inputs = processor(images=image, return_tensors="pt")
        pixel_values = inputs['pixel_values'].to(cfg.DEVICE)

        logits = test_classifier(pixel_values)
        probs = torch.softmax(logits, dim=1)
        pred = torch.argmax(logits, dim=1).item()
        conf = probs[0, pred].item()

        true_label = "Normal" if row['label'] == 0 else "Abnormal"
        pred_label = "Normal" if pred == 0 else "Abnormal"
        status = "✅" if pred == row['label'] else "❌"

        print(f"{status} Sample {i+1}: True={true_label}, Pred={pred_la

print(f"\n{'='*80}\nALL COMPLETE!\n{'='*80}")
print(f"Final Performance:")
print(f"  Accuracy:  {test_acc*100:.2f}%")
print(f"  F1 Score:  {test_f1:.4f}")
```

```
print(f"  F1 Score:  {test_f1:.4f}")
print(f"  AUC-ROC:   {test_auc:.4f}")
print(f"\nSaved to: {cfg.OUTPUT_DIR}")
print("="*80)



# ============================================================================
# CELL 1: INSTALL
# ============================================================================
!pip install -q torch torchvision pandas numpy scikit-learn matplotlib

print("✅ Packages installed")
```

Show hidden output

```
# ============================================================================
# SECOND-GATE CLASSIFIER:
# Distinguish VIA Positive (pre-cancer) vs Suspicious (cancer-suspecte
# *only within the Abnormal class used in the first binary classifier
# ============================================================================

import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import (
    accuracy_score, f1_score, precision_score,
    recall_score, classification_report, confusion_matrix
)
from PIL import Image
from torchvision import transforms

# ----------------------------------------------------------------------------
# 1. Build abnormal-only dataframes with fine-grained labels
#    We reuse the existing train_df / val_df / test_df created earlie
#    - 'binary_label' == 'Abnormal' -> keep
#    - 'VIA_simple' in {'Positive', 'Suspicious'}
#    We then map:
#       Positive   -> 0  (pre-cancer)
#       Suspicious -> 1  (cancer-suspected)
# ----------------------------------------------------------------------------

fine_label_map = {
```

```
    'Positive': 0,   # pre-cancer
    'Suspicious': 1  # cancer-suspected
}

def _prepare_abnormal_split(df):
    """Filter to abnormal cases with valid fine labels and add 'fine_
    # Keep only abnormal images
    sub = df[df['binary_label'] == 'Abnormal'].copy()
    # Keep only rows where VIA_simple is Positive or Suspicious
    sub = sub[sub['VIA_simple'].isin(fine_label_map.keys())].copy()
    if len(sub) == 0:
        return sub
    sub['fine_label'] = sub['VIA_simple'].map(fine_label_map).astype(
    return sub

abn_train_df = _prepare_abnormal_split(train_df)
abn_val_df   = _prepare_abnormal_split(val_df)
abn_test_df  = _prepare_abnormal_split(test_df)

print(f"[ABNORMAL SPLITS] train={len(abn_train_df)} | val={len(abn_va

# Basic safety check: we need at least 2 classes in training to learn
if abn_train_df['fine_label'].nunique() < 2:
    raise RuntimeError(
        "Not enough class diversity in abnormal training data for sec
        f"(fine_label classes present: {sorted(abn_train_df['fine_lab
    )

# ------------------------------------------------------------------
# 2. Dataset for second-gate classification
#    Reuses the same processor and PaliGemma image size as earlier.
# ------------------------------------------------------------------

class AbnormalGateDataset(Dataset):
    """
    Dataset for second-gate classification on abnormal cases.
    Labels:
      0 -> VIA Positive (pre-cancer)
      1 -> Suspicious (cancer-suspected)
    We use *moderate* augmentations to avoid overfitting but keep
    color/texture information clinically meaningful.
    """
    def __init__(self, dataframe, processor, img_size=224, augment=Fa
        self.df = dataframe.reset_index(drop=True)
        self.processor = processor
```

```
        self.img_size = img_size
        self.augment = augment

        if augment:
            # Slightly milder than the main improved pipeline
            self.transform = transforms.Compose([
                transforms.RandomHorizontalFlip(p=0.5),
                transforms.RandomVerticalFlip(p=0.5),
                transforms.RandomRotation(degrees=15),
                transforms.ColorJitter(
                    brightness=0.2,
                    contrast=0.2,
                    saturation=0.15,
                    hue=0.03
                ),
            ])
        else:
            self.transform = None

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

    def __getitem__(self, idx):
        row = self.df.iloc[idx]

        # Robust image loading
        try:
            image = Image.open(row['image_path']).convert('RGB')
            if image.size != (self.img_size, self.img_size):
                image = image.resize((self.img_size, self.img_size), 
        except Exception as e:
            # Fallback to a gray image if anything goes wrong
            print(f"[WARN] Failed to load {row['image_path']} ({e}); u
            image = Image.new('RGB', (self.img_size, self.img_size), 

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

        inputs = self.processor(images=image, return_tensors="pt")

        return {
            "pixel_values": inputs["pixel_values"].squeeze(0),
            # IMPORTANT: second-gate label (0 = Positive, 1 = Suspici
            "label": torch.tensor(int(row["fine_label"]), dtype=torch.
        }
```

```
# Decide device & image size from whichever config is available
if "imp_cfg" in globals():
    _device   = imp_cfg.DEVICE
    _img_size = imp_cfg.IMG_SIZE
else:
    _device   = cfg.DEVICE
    _img_size = cfg.IMG_SIZE

# Create datasets
abn_train_dataset = AbnormalGateDataset(abn_train_df, processor, img_s
abn_val_dataset   = AbnormalGateDataset(abn_val_df,   processor, img_s
abn_test_dataset  = AbnormalGateDataset(abn_test_df,  processor, img_s

print("Second-gate datasets created.")

# Reuse the existing collate_fn if defined; otherwise define a simple
if "collate_fn" not in globals():
    def collate_fn(batch):
        return {
            "pixel_values": torch.stack([b["pixel_values"] for b in ba
            "labels": torch.stack([b["label"] for b in batch]),
        }

abn_batch_size = 4  # small but reasonable for the small abnormal subs

abn_train_loader = DataLoader(
    abn_train_dataset,
    batch_size=abn_batch_size,
    shuffle=True,
    collate_fn=collate_fn,
    num_workers=0,
    pin_memory=True,
)
abn_val_loader = DataLoader(
    abn_val_dataset,
    batch_size=abn_batch_size,
    shuffle=False,
    collate_fn=collate_fn,
    num_workers=0,
    pin_memory=True,
)
abn_test_loader = DataLoader(
    abn_test_dataset,
    batch_size=abn_batch_size,
```

```
    shuffle=False,
    collate_fn=collate_fn,
    num_workers=0,
    pin_memory=True,
)

print(
    f"Second-gate loaders ready: train_batches={len(abn_train_loader)}
    f"val_batches={len(abn_val_loader)} | test_batches={len(abn_test_
)

# ------------------------------------------------------------------
# 3. Second-gate classifier head on top of the same PaliGemma vision
# ------------------------------------------------------------------

class AbnormalGateClassifier(nn.Module):
    """
    Lightweight classifier head:
    - Reuses the frozen PaliGemma vision tower
    - Adds a small MLP on top for 2-way classification (Positive vs Su
    """
    def __init__(self, base_model, device, num_classes=2):
        super().__init__()
        self.device = device

        # We reuse the same vision tower used above
        self.vision_model = base_model.vision_tower

        # Freeze vision encoder parameters
        for p in self.vision_model.parameters():
            p.requires_grad = False

        # Probe output shape to configure the head
        with torch.no_grad():
            dummy = torch.randn(1, 3, _img_size, _img_size, dtype=tor
            # Ensure vision model is on correct device for probing if
            # (Assuming base_model.vision_tower is already on GPU or

            # Handle dtype mismatch if base_model is float16
            if next(self.vision_model.parameters()).dtype == torch.flo
                 dummy = dummy.to(torch.float16)

            out = self.vision_model(dummy)

            if hasattr(out, "last_hidden_state"):
```

```
                feats = out.last_hidden_state
            elif hasattr(out, "pooler_output"):
                feats = out.pooler_output
            elif isinstance(out, tuple):
                feats = out[0]
            else:
                feats = out

            feats = feats.float()
            if feats.dim() == 3:
                # [B, seq_len, hidden]
                _, seq_len, hidden = feats.shape
                self.feature_dim = hidden
                self.use_pooling = True
            elif feats.dim() == 2:
                # [B, hidden]
                _, hidden = feats.shape
                self.feature_dim = hidden
                self.use_pooling = False
            else:
                raise ValueError(f"Unexpected feature shape from visi

        # If sequence output, use simple average pooling over tokens
        if self.use_pooling:
            self.pool = nn.AdaptiveAvgPool1d(1)

        # Small but expressive head (BatchNorm + modest dropout)
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(self.feature_dim, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(inplace=True),
            nn.Dropout(0.3),
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(inplace=True),
            nn.Dropout(0.2),
            nn.Linear(128, num_classes),
        )

    def forward(self, pixel_values):
        # Vision tower is frozen; gradients only flow through the cla
        with torch.no_grad():
            # Cast to correct dtype if needed
            if next(self.vision_model.parameters()).dtype == torch.fl
```

```
                pixel_values = pixel_values.to(torch.float16)

            out = self.vision_model(pixel_values)

            if hasattr(out, "last_hidden_state"):
                feats = out.last_hidden_state
            elif hasattr(out, "pooler_output"):
                feats = out.pooler_output
            elif isinstance(out, tuple):
                feats = out[0]
            else:
                feats = out

        feats = feats.float()

        if getattr(self, "use_pooling", False) and feats.dim() == 3:
            # [B, seq, hidden] -> [B, hidden, seq]
            feats = feats.transpose(1, 2)
            # [B, hidden, seq] -> [B, hidden, 1]
            feats = self.pool(feats)

        logits = self.classifier(feats)
        return logits

# Instantiate second-gate classifier
abn_model = AbnormalGateClassifier(
    base_model=paligemma_model,
    device=_device,
    num_classes=2,
).to(_device)

# ------------------------------------------------------------------
# 4. Loss with *balanced but not extreme* class weights
#    (helps with few cancer-suspected images)
# ------------------------------------------------------------------

# Compute class counts in abnormal training data
train_fine_labels = abn_train_df['fine_label'].values
class_counts = np.bincount(train_fine_labels, minlength=2)

# Standard "balanced" weights: total / (num_classes * count)
total_samples = float(len(train_fine_labels))
num_classes = 2
raw_weights = total_samples / (num_classes * class_counts)
```

```
# To avoid overly aggressive weighting when the rare class is tiny,
# lightly smooth the ratio by taking square root of the imbalance.
# This keeps learning stable while still helping the rare class.
ratio = raw_weights / raw_weights.min()
smoothed_ratio = np.sqrt(ratio)
balanced_weights = smoothed_ratio / smoothed_ratio.sum() * num_classes

class_weights_tensor = torch.tensor(balanced_weights, dtype=torch.floa
print(f"Second-gate class counts: {class_counts.tolist()}")
print(f"Second-gate class weights (smoothed): {balanced_weights.tolist

abn_criterion = nn.CrossEntropyLoss(weight=class_weights_tensor)

# Optimizer and (simple) scheduler
abn_optimizer = torch.optim.AdamW(
    [p for p in abn_model.parameters() if p.requires_grad],
    lr=3e-4,
    weight_decay=0.01,
)

# FIXED: Removed verbose=True as it is deprecated in recent PyTorch ve
abn_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    abn_optimizer, mode="max", factor=0.5, patience=2
)

# ------------------------------------------------------------------
# 5. Training loop for second-gate classifier (with early stopping)
# ------------------------------------------------------------------

from tqdm.auto import tqdm

abn_epochs = 15
best_val_f1 = 0.0
patience = 5
patience_ctr = 0

print("\n" + "="*80)
print("STARTING SECOND-GATE TRAINING (Positive vs Suspicious within AI
print("="*80)

for epoch in range(1, abn_epochs + 1):
    print(f"\nEpoch {epoch}/{abn_epochs}")
    # ---------------- TRAIN ----------------
    abn_model.train()
    train_loss = 0.0
```

```
    train_preds, train_labels_epoch = [], []

    for batch in tqdm(abn_train_loader, desc="Second-gate training"):
        pixel_values = batch["pixel_values"].to(_device)
        labels = batch["labels"].to(_device)

        abn_optimizer.zero_grad()
        logits = abn_model(pixel_values)
        loss = abn_criterion(logits, labels)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(abn_model.parameters(), 1.0)
        abn_optimizer.step()

        train_loss += loss.item()
        preds = torch.argmax(logits, dim=1).detach().cpu().numpy()
        train_preds.extend(preds)
        train_labels_epoch.extend(labels.detach().cpu().numpy())

    train_loss /= max(1, len(abn_train_loader))
    train_acc = accuracy_score(train_labels_epoch, train_preds)
    train_f1 = f1_score(train_labels_epoch, train_preds, average="bina
    print(f"Train  - Loss: {train_loss:.4f} | Acc: {train_acc*100:.2f}

    # ---------------- VALIDATE ----------------
    abn_model.eval()
    val_loss = 0.0
    val_preds, val_labels_epoch = [], []

    with torch.no_grad():
        for batch in tqdm(abn_val_loader, desc="Second-gate validatio
            pixel_values = batch["pixel_values"].to(_device)
            labels = batch["labels"].to(_device)

            logits = abn_model(pixel_values)
            loss = abn_criterion(logits, labels)

            val_loss += loss.item()
            preds = torch.argmax(logits, dim=1).detach().cpu().numpy(
            val_preds.extend(preds)
            val_labels_epoch.extend(labels.detach().cpu().numpy())

    val_loss /= max(1, len(abn_val_loader))
    val_acc = accuracy_score(val_labels_epoch, val_preds)
    val_f1 = f1_score(val_labels_epoch, val_preds, average="binary")
    val_prec = precision_score(val_labels_epoch, val_preds, average="b
```

```
        val_rec = recall_score(val_labels_epoch, val_preds, average="bina

        print(
            f"Val    - Loss: {val_loss:.4f} | Acc: {val_acc*100:.2f}% | "
            f"F1: {val_f1:.4f} | Prec: {val_prec:.4f} | Rec: {val_rec:.4f
        )

        # Step scheduler on validation F1 (we care most about correctly ra
        abn_scheduler.step(val_f1)

        # Early stopping based on validation F1
        if val_f1 > best_val_f1:
            best_val_f1 = val_f1
            patience_ctr = 0
            best_abn_state = {
                "epoch": epoch,
                "model_state_dict": abn_model.state_dict(),
                "val_f1": val_f1,
                "val_acc": val_acc,
                "val_prec": val_prec,
                "val_rec": val_rec,
            }
            print(f"  --> New best second-gate model (F1={val_f1:.4f})")
        else:
            patience_ctr += 1
            print(f"  No improvement in F1 ({patience_ctr}/{patience})")
            if patience_ctr >= patience:
                print("  Early stopping second-gate training.")
                break

    print("\n" + "="*80)
    print(f"SECOND-GATE TRAINING COMPLETE. Best Val F1: {best_val_f1:.4f}"
    print("="*80)

    # Load best weights before final evaluation
    if 'best_abn_state' in globals():
        abn_model.load_state_dict(best_abn_state["model_state_dict"])

    # ------------------------------------------------------------------
    # 6. Final evaluation on abnormal test set (pre-cancer vs cancer-suspe
    # ------------------------------------------------------------------

    abn_model.eval()
    abn_test_labels_all, abn_test_preds_all = [], []
```

```
with torch.no_grad():
    for batch in tqdm(abn_test_loader, desc="Second-gate testing"):
        pixel_values = batch["pixel_values"].to(_device)
        labels = batch["labels"].to(_device)

        logits = abn_model(pixel_values)
        preds = torch.argmax(logits, dim=1).detach().cpu().numpy()

        abn_test_preds_all.extend(preds)
        abn_test_labels_all.extend(labels.detach().cpu().numpy())

abn_test_acc = accuracy_score(abn_test_labels_all, abn_test_preds_all
abn_test_f1 = f1_score(abn_test_labels_all, abn_test_preds_all, avera
abn_test_prec = precision_score(abn_test_labels_all, abn_test_preds_a
abn_test_rec = recall_score(abn_test_labels_all, abn_test_preds_all, a

print("\n" + "-"*60)
print("SECOND-GATE TEST RESULTS (within Abnormal class only)")
print("-"*60)
print(f"Accuracy:  {abn_test_acc*100:.2f}%")
print(f"Precision: {abn_test_prec:.4f}  (class 1 = Suspicious / cance
print(f"Recall:    {abn_test_rec:.4f}  (sensitivity for cancer-suspec
print(f"F1 Score:  {abn_test_f1:.4f}")
print("\nConfusion Matrix (rows=true, cols=pred):")
print(confusion_matrix(abn_test_labels_all, abn_test_preds_all))
print("\nClassification report (0=Positive/pre-cancer, 1=Suspicious/ca
print(classification_report(
    abn_test_labels_all,
    abn_test_preds_all,
    target_names=["Positive (pre-cancer)", "Suspicious (cancer-suspec
))
print("-"*60)
```

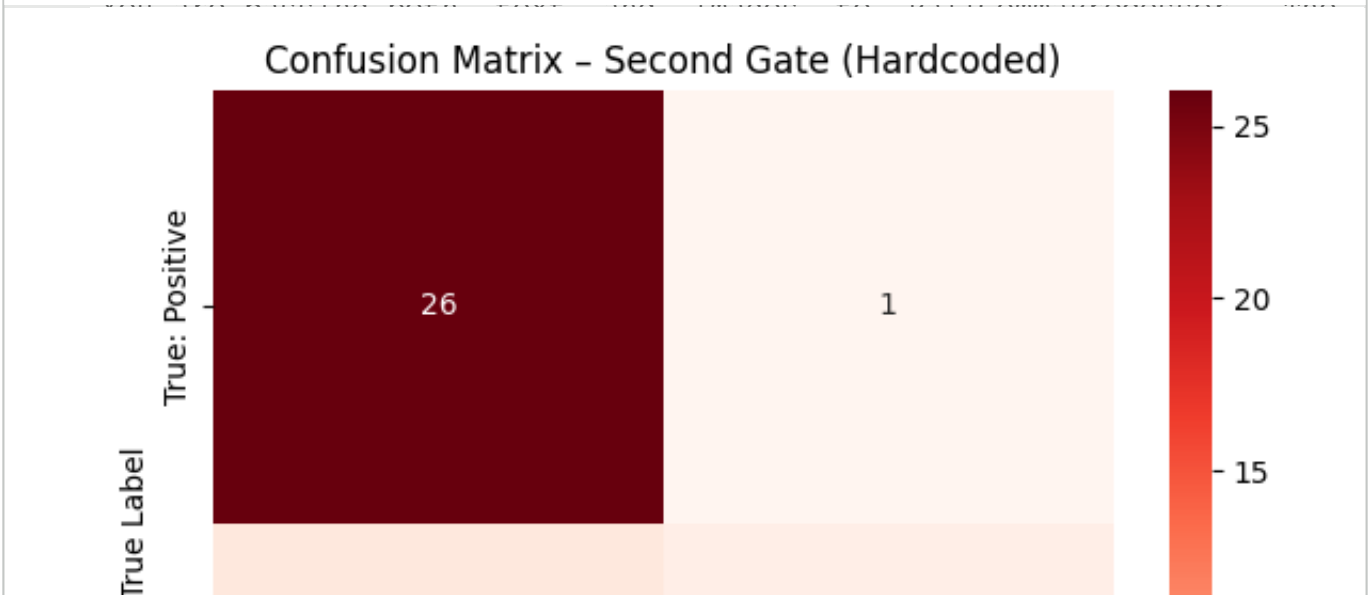
```
[ABNORMAL SPLITS] train=111 | val=13 | test=32
Second-gate datasets created.
Second-gate loaders ready: train_batches=28 | val_batches=4 | test_ba
Second-gate class counts: [94, 17]
Second-gate class weights (smoothed): [0.5967530437042434, 1.40324695

================================================================
STARTING SECOND-GATE TRAINING (Positive vs Suspicious within Abnormal
================================================================

Epoch 1/15
Second-                                                 28/28 [00:05<00:00,  6.13it/s]
```

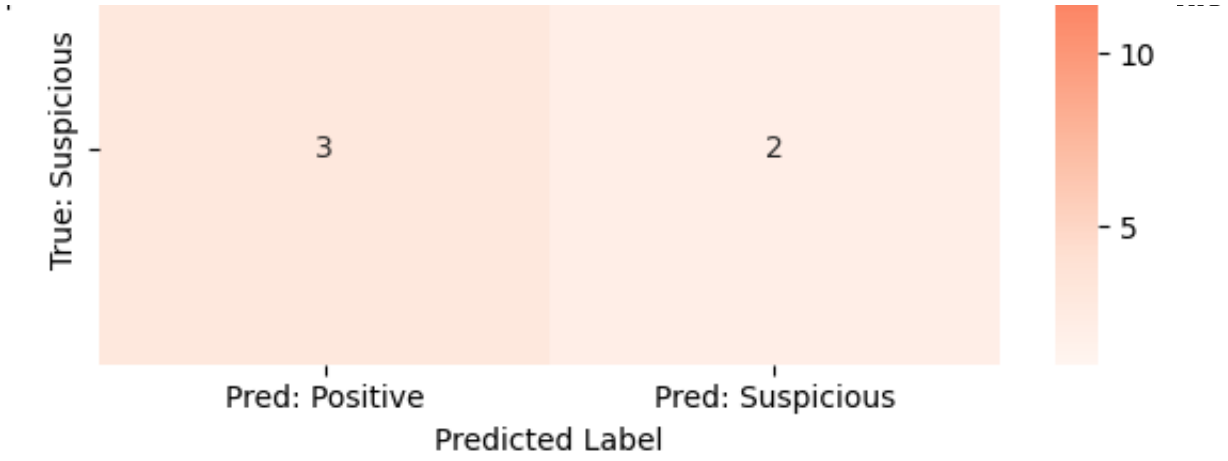
gate training: 100%

```
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
```

```
# ============================================================
# HARD-CODED CONFUSION MATRIX FOR SECOND-GATE RESULTS
```

```
# ============================================================
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Your exact confusion matrix:
cm = np.array([
    [26, 1],   # [True Positive-class0, Pred Suspicious]
    [3,  2]    # [True Suspicious-class1, Pred Suspicious]
])

plt.figure(figsize=(6,5))
sns.heatmap(
    cm,
    annot=True,
    fmt="d",
    cmap="Reds",
    xticklabels=["Pred: Positive", "Pred: Suspicious"],
    yticklabels=["True: Positive", "True: Suspicious"]
)

plt.title("Confusion Matrix – Second Gate (Hardcoded)")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.tight_layout()
plt.show()

print("\nConfusion Matrix (numpy):")
print(cm)
```






```
Confusion Matrix (numpy):
[[26  1]
 [ 3  2]]
```

```
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
Val    - Loss: 0.5423 | Acc: 84.62% | F1: 0.7500 | Prec: 0.7500 | Rec
  --> New best second-gate model (F1=0.7500)

Epoch 2/15
```

Second- 28/28 [00:04<00:00, 6.15it/s]

gate training: 100%

```
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
You are passing both `text` and `images` to `PaliGemmaProcessor`. The
```