Documentation of the Pipeline

This documentation describes the functionality, purpose, and detailed step-by-step operations of critical components in the automated image labeling and model training pipeline:

1. **preprocessing.py** — preprocessing and test data augmentation
2. **preprocessing_expanded.py** – expanded preprocessing (CLIP cosine similarity and SAM2 crop)
3. **sam2_mask_generator.py** — mask generation with CLIP filtering
4. **autolabel_and_train.py** — automatic pseudo-labeling and training initiation
5. **classified_crops.py** – automated object detection and classification
6. **sam2_surgical_mask.py** – precise surgical instrument mask generation
7. **generate_masks_with_sam.py** – binary segmentation masks
8. **Model LoRA 1:**
    a. **inference.py** – Inference and Multi-Object Detection Using SAM2 + ViT+LoRA
    b. **model.py** – Fine-Tuning ViT with LoRA for Instrument Classification
9. **Model LoRA 2:**
    a. **inference.py** – semantic segmentation
    b. **surgical_instrument_segmentation.py** – semantic segmentation model (SegFormer with LoRA)
10. **Model YOLO LoRA** – YOLOv8 Custom Training Pipeline

Detailed documentation

1. **preprocessing.py** - Preprocessing Raw Surgical Images
    a. Overview
    This script prepares raw surgical instrument images for deep learning models by resizing, normalizing, adjusting color properties, and generating augmented test images through rotation. In other words, it executes the data preprocessing pipeline, preparing images for training and generating a test dataset through rotation-based augmentation.
    b. Technologies Used
        i. OpenCV (cv2): For image loading, resizing, processing, and saving.
        ii. NumPy: For pixel-level numerical operations.
        iii. Standardization (Normalization): Using mean and standard deviation values from ImageNet to align the dataset with pretrained CNN expectations.
        iv. Augmentation (Rotation): Improves model generalization by rotating images at multiple fixed angles.
    c. Key Parameters
        i. TARGET_SIZE = (224, 224) # Resizes all images to 224x224 (standard for CNNs like ResNet)
        ii. ROTATION_ANGLES = [-15, -10, -5, 5, 10, 15] # Used for test set augmentation
        iii. MEAN = np.array([0.485, 0.456, 0.406]) # ImageNet channel-wise mean (R, G, B)
        iv. STD = np.array([0.229, 0.224, 0.225]) # ImageNet channel-wise std dev (R, G, B)
        v. DATA_DIR #Directory of original unprocessed image data

vi. PROCESSED_DIR #Output location for standardized training-ready images
vii. TEST_DIR #Destination for rotated images used in testing
viii. ROTATION_ANGLES #List of angles (e.g., [90, 180, 270]) to rotate each image by

d. Directory Structure
   i. Project/
   ```
   │
   ├── Data/          # Raw images (input)
   ├── Processed/     # Preprocessed output images
   └── Test/          # Augmented rotated images
   ```

e. Steps:
   i. Step 1: Preprocess Original Images → PROCESSED_DIR
      - Function: preprocess_image(src_path, out_dir, out_name)
      - Why: Standardizes raw image data by resizing, converting color channels, and applying normalization to reduce input noise and improve model performance.
      - Input: All image files (.jpg, .png, .jpeg) in DATA_DIR
      - Output: Clean .jpg images saved in PROCESSED_DIR
   ii. Step 2: Generate Rotated Test Set → TEST_DIR
      - Function:
      ```
      for angle in ROTATION_ANGLES:
          rotated = rotate(img, angle)
          save(rotated, TEST_DIR)
      ```
      - Why: Enhances the model evaluation set by simulating real-world orientation changes. Helps test the model's ability to generalize.
      - Mechanism: Uses OpenCV's warpAffine() with reflection border handling to avoid cropping

f. Function: adjust_contrast_saturation(img, alpha=1.1, beta=0, sat_scale=1.1)
   i. Purpose: Enhance image visibility by adjusting contrast, brightness, and saturation.
   ii. Arguments:
      - img (np.ndarray): Input image.
      - alpha (float): Contrast multiplier.
      - beta (int): Brightness adjustment.
      - sat_scale (float): Saturation multiplier.
   iii. Returns: np.ndarray — Modified image with adjusted properties.
   iv. How It Works:
      - Adjusts contrast/brightness via cv2.convertScaleAbs().
      - Converts to HSV to scale saturation.
      - Converts back to BGR for further processing.

g. Function: preprocess_image(src_path, dst_dir, base_name)
   i. Purpose: Applies full preprocessing pipeline to one image. Loads an image, resizes and normalizes it, and saves it to the target folder.
   ii. Steps:

- Read image from src_path.
- Resize to TARGET_SIZE using bilinear interpolation.
- Enhance contrast and saturation.
- Normalize pixel values to [0,1] and standardize using ImageNet stats.
- De-normalize for saving (convert back to uint8).
- Save preprocessed image in dst_dir.

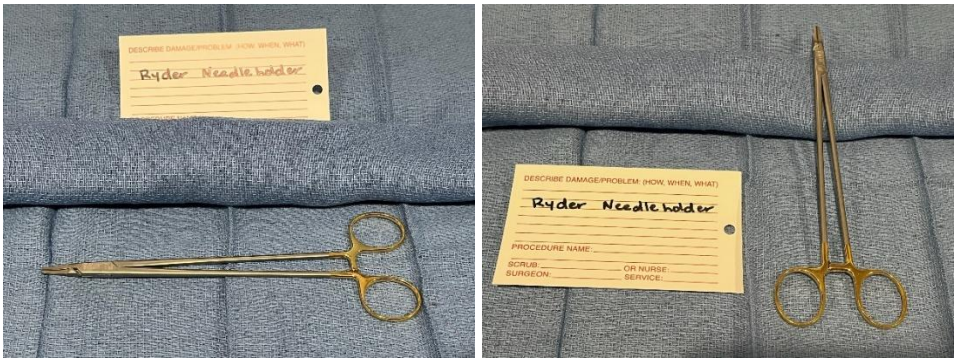
iii. Arguments:

- src_path (str): Path to input image.
- dst_dir (str): Destination directory.
- base_name (str): Output filename without extension.

iv. Returns: None (saves image to disk)

h. Results

i. RAW Image Example (Ryder Needle Holder):



ii. Processed Image Example (Ryder Needle Holder):

iii. Test Image Example (Ryder Needle Holder):

2. **preprocessing_expanded.py** – Expanded Preprocessing (CLIP Cosine Similarity and SAM2 Crop)
    a. Overview
        i. Same as preprocessing.py, but CLIP cosine similarity algorithm and SAM2 crop technic are added to enhance the quality of labeling and cropping.

3. **sam2_mask_generator.py** - Mask Generation with SAM2 + CLIP
    a. Overview
    This script uses Meta's Segment Anything Model (SAM2) to automatically generate object masks from images, and filters them using OpenAI's CLIP zero-shot model based on semantic similarity. It uses a trained model with its vector representation to make predictions on test images, filters them by confidence, organizes them into folders by predicted class, and logs the output.

b. Technologies Used
    i. Segment Anything Model (SAM2): State-of-the-art zero-shot image segmentation model from Meta.
    ii. CLIP (Contrastive Language–Image Pretraining): A vision-language model from OpenAI to match image content with text prompts.
    iii. OpenCV & PIL: Image loading and manipulation.
    iv. Torch & NumPy: Model execution and matrix operations.
c. Command-Line Arguments
    i. --image: Input image path (required)
    ii. –checkpoint: Path to the SAM2 .pth model checkpoint (required)
    iii. --model_type: SAM2 backbone (vit_h / vit_l / vit_b)
    iv. --output_dir: Directory to save output masks (default: masks_output)
    v. --threshold: CLIP similarity threshold for filtering (default: 0.3)
    vi. --visualize: Whether to display masks visually using matplotlib
d. Function: parse_args()
    i. Purpose: Parses CLI arguments for script flexibility.
e. Function: load_sam(checkpoint_path, model_type)
    i. Purpose: Loads SAM2 model from given checkpoint and model type.
    ii. Arguments:
        - checkpoint_path (str): Path to .pth file.
        - model_type (str): One of "vit_h", "vit_l", "vit_b".
    iii. Returns: A loaded SAM model.
f. Function: setup_clip(device)
    i. Purpose: Loads CLIP model (ViT-B/32) and its preprocessing pipeline onto the given device (CPU/GPU).
    ii. Returns:
        - model: CLIP model.
        - preprocess: Preprocessing function for CLIP.
g. Stage of masks generation
    i. Purpose: Generates a list of segmentation masks using SAM2.
    ii. Arguments:
        - sam: Loaded SAM model.
        - image (np.ndarray): RGB input image.
    iii. Returns:
        - List of mask dictionaries containing segmentation maps.
h. Filtering with CLIP (in-progress section)
    i. Crop each mask from the original image.
    ii. Run CLIP inference for each crop.
    iii. Compare similarity to a text description (e.g., "surgical instrument").
    iv. Retain only masks with similarity score > threshold.

    This will improve mask quality by ensuring semantic alignment, removing false positives.
i. Function: save_masks(masks, output_dir)
    i. Purpose: Saves each segmentation mask as a binary .png.

ii. Arguments:
- masks: List of dictionaries from SAM.
- output_dir: Directory to save masks.

iii. Implementation:
- Converts boolean masks to uint8 images (0 or 255).
- Saves with filenames like mask_000.png.

j. Function: visualize_masks(image, masks)
i. Purpose: Overlay and visualize the segmentation masks on the image.
ii. Implementation:
- Sorts masks by area (largest first).
- Applies random color overlays with alpha blending using matplotlib.

k. Output
i. High-quality binary masks in output_dir/.
ii. (Optional) Visualizations of overlaid masks.
iii. Future support for CLIP-filtered masks.

4. **autolabel_and_train.py** - Automatic Pseudo-Labeling and Training Initiation
a. Overview
This script implements a two-stage segmentation pipeline:
i. Auto-labeling masks generated by SAM (Segment Anything Model) using OpenAI's CLIP for classification.
ii. Training a Mask R-CNN instance segmentation model using PyTorch's torchvision on the labeled dataset.

b. Stages:
i. Stage 1: Auto-labeling with CLIP
- Function: autolabel_masks(...)
a. This step classifies each mask using CLIP image-text similarity.
- Purpose:
a. Automatically classify segmentation masks into instrument categories.
- Workflow:
a. Load CLIP model:
i. Load ViT-B/32 with clip.load().
ii. Encode provided class names into text features.
b. Setup folder structure:
i. Create a subfolder in output_dir for each class.
c. Loop through mask files:
i. Skip non-PNG files.
ii. Derive image name from the mask filename (e.g., image001_000.png → image001.jpg).
iii. Open the original image and crop it to the bounding box of the mask.

d. Use CLIP for classification:
    i. Preprocess the image crop.
    ii. Encode the image with CLIP.
    iii. Compute similarity with class text embeddings.
    iv. If similarity > threshold, copy the mask to the predicted class folder.

- Parameters:
    a. masks_dir: Folder containing binary SAM-generated masks.
    b. images_dir: Folder with full original images (in .jpg or .png).
    c. output_dir: Where the labeled masks will be saved by class.
    d. classes: List of instrument class names (e.g., forceps, scalpel).
    e. threshold: Confidence threshold for assigning a label.
    f. device: "cuda" or "cpu".

ii. Stage 2: Dataset Loader

- Class: InstrumentDataset
    a. This class prepares the dataset for Mask R-CNN training.
- Folder Structure Required:
    a. dataset_root/

```
├── images/
│   └── image001.jpg
├── masks/
│   ├── DeBakey_forceps/
│   │   └── image001_000.png
│   ├── curved_mosquito/
│   │   └── image001_001.png
```

- Functionality:
    a. Loads each image.
    b. Loads all matching masks across class subfolders for that image.
    c. For each mask:
        i. Computes a bounding box.
        ii. Converts the mask to binary.
        iii. Stores class label, mask, and box.
- Returns:
    a. img: Transformed image tensor.
    b. target: Dictionary with:
        i. "boxes": bounding boxes
        ii. "labels": integer class labels
        iii. "masks": binary segmentation masks

iii. Stage 3: Model Training

- Function: train_model(...)
    a. Trains a Mask R-CNN segmentation model using the dataset.
- Workflow:
    a. Dataset:

i. Load InstrumentDataset from dataset_root.

b. Model Setup:

i. Load maskrcnn_resnet50_fpn with the correct number of classes (len(classes) + 1 to include background).

ii. Move model to target device.

c. Optimizer:

i. Use SGD with momentum and weight decay.

d. Training Loop:

i. For each epoch:

1. Forward pass each batch.
2. Compute total loss across outputs.
3. Backpropagate and update weights.
4. Print epoch loss.

e. Save Model:

i. Model is saved as maskrcnn_checkpoint.pth inside dataset_root.

- Parameters:
  a. dataset_root: Path to training data (with images/ and masks/).
  b. classes: List of instrument classes (without background).
  c. epochs: Number of training epochs.
  d. batch_size: Training batch size.
  e. lr: Learning rate.
  f. device: "cuda" or "cpu".

c. Command-Line Interface

i. Function: main()

- Provides a command-line interface with two subcommands: autolabel and train.

ii. Usage Examples:

- Auto-labeling masks:

```
python autolabel_and_train.py autolabel `
  --masks_dir sam_output/ `
  --images_dir dataset/images/ `
  --output_dir dataset/masks/ `
  --classes right_angle_clamp curved_mosquito DeBakey_forceps
angled_bulldog_clamp `
  --threshold 0.3 `
  --device cuda
```

- Training model:

```
python autolabel_and_train.py train `
  --dataset_root dataset/ `
  --classes right_angle_clamp curved_mosquito DeBakey_forceps
angled_bulldog_clamp `
  --epochs 15 `
  --batch_size 4 `
```

```
--lr 0.005 `
--device cuda
```

d. Notes
   i. Make sure masks are binary (non-zero foreground).
   ii. Ensure naming consistency between mask files and image files.
   iii. Add __background__ automatically to class list during training.

5. **classified_crops.py** - automated object detection and classification
   a. Overview
      It is a Python script that performs automated object detection and classification on a single surgical image. It combines two models:
      i. SAM (Segment Anything Model) from Meta for object proposal generation.
      ii. CLIP (Contrastive Language–Image Pretraining) from OpenAI for zero-shot classification.

      The script is designed to:
      i. Load a surgical image.
      ii. Use SAM to extract bounding box proposals of objects.
      iii. Filter and optionally visualize these proposals.
      iv. Classify the cropped image regions using CLIP with a pre-defined set of candidate labels.
      v. Save the cropped, classified images to disk in a structured output folder.
   b. Requirements
      i. pip install torch torchvision torchaudio opencv-python matplotlib Pillow transformers
      ii. pip install git+https://github.com/facebookresearch/segment-anything.git
   c. Project Structure
      i. classified_crops.py # Main script
      ii. compressed.jpeg # Input image (or replace with your own)
      iii. sam_vit_h_4b8939.pth # SAM model checkpoint
      iv. classified_crops_matched_params/
      v. └── label/
      vi. └── crop_000.png # Saved classified object crops
   d. Configuration
      i. SAM (Segment Anything Model)

```
SAM_IMAGE_INPUT = "compressed.jpeg"
SAM_CHECKPOINT = "sam_vit_h_4b8939.pth"
SAM_MODEL_TYPE = "vit_h"
```

      ii. SAM Parameters (aligned to baseline script)

```
SAM_POINTS_PER_SIDE = 32
SAM_PRED_IOU_THRESH = 0.9
SAM_STABILITY_SCORE_THRESH = 0.92
SAM_MIN_MASK_REGION_AREA = 50000
SAM_BOX_NMS_THRESH = 0.7
```

iii. Filtering Bounding Boxes
APPLY_FRACTIONAL_AREA_FILTER = True
MIN_BBOX_AREA_FRACTION = 0.03
MAX_BBOX_AREA_FRACTION = 0.15

iv. CLIP Classification
CLIP_MODEL_CHECKPOINT = "openai/clip-vit-large-patch14"
CANDIDATE_LABELS = ["Scalpel", "Forceps", ..., "Object on surgical drape"]

e. Processing Pipeline
   i. Load and Verify Image
      - Reads the input image using OpenCV.
      - Converts BGR to RGB format for model compatibility.
   ii. Load SAM
      - Loads the selected SAM model to the specified device.
      - Applies SamAutomaticMaskGenerator to generate object masks and bounding boxes.
   iii. Filter Bounding Boxes
      - Optionally filters bounding boxes by area fraction relative to the image.
   iv. Crop & Classify with CLIP
      - Extracts the bounding box crops from the original image.
      - Feeds each crop into CLIP's zero-shot classification pipeline.
      - Assigns the top-1 label from the defined candidate label list.
   v. Save Results
      - Crops are saved in folders named after their predicted label.
      - Filenames are sanitized and uniquely numbered.
   vi. Optional Visualization
      - Bounding boxes can be visualized on the original image using matplotlib and OpenCV for debugging or inspection.

f. Key Functions
   i. generate_sam_bounding_box_proposals(...)
      - Inputs: image path, SAM parameters
      - Outputs: list of [x_min, y_min, x_max, y_max, area], original image, raw SAM masks
      - Handles all logic for SAM inference and postprocessing.
   ii. show_boxes_on_image(...)
      - Visualizes bounding boxes and their areas on top of the original image.
      - Useful for manual verification.
   iii. sanitize_filename(...)
      - Replaces spaces and invalid characters in label names for safe file naming.

g. Example Run
   i. python classified_crops.py
   ii. Ensure the following are in place:
      - compressed.jpeg exists

- sam_vit_h_4b8939.pth is downloaded and located properly
- Proper Python environment is activated

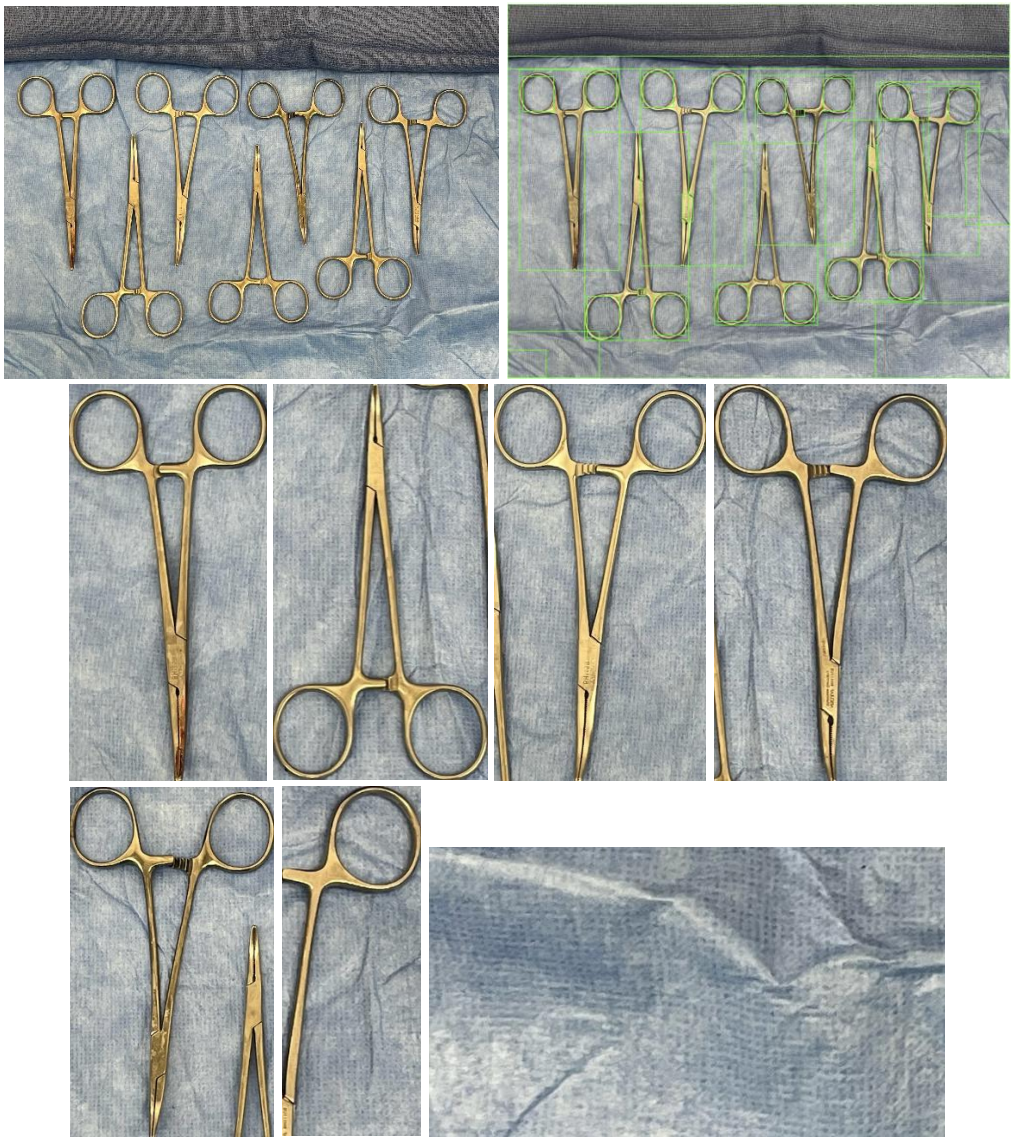
h. Notes

i. For custom use cases, you may update CANDIDATE_LABELS with relevant terms for your own dataset.

ii. The SAM parameters provided are tuned for large surgical instruments. Adjust SAM_MIN_MASK_REGION_AREA if working with smaller items.

iii. If many false positives occur, increase the SAM_PRED_IOU_THRESH or SAM_STABILITY_SCORE_THRESH.

i. Results



6. **sam2_surgical_mask.py** – Precise Surgical Instrument Mask Generation
    a. Overview
    This script utilizes the Segment Anything Model v2 (SAMv2) to segment surgical instruments from an input image. It identifies the most relevant mask (based on area), crops the bottom (typically where instruments lie), and exports a transparent image preserving only the instrument area.
    b. Overview
    sam2_surgical_mask.py provides a full pipeline to:
        i. Load an input surgical image.
        ii. Initialize the SAMv2 model using a specified checkpoint.
        iii. Automatically generate segmentation masks.
        iv. Select the most prominent surgical instrument mask.
        v. Crop the image to retain only the bottom portion (where instruments are likely to be).
        vi. Apply the mask to isolate the instrument with a transparent background.
        vii. Save the result as a .png image.
    c. Requirements
        i. Python 3.7+
        ii. PyTorch
        iii. segment_anything package (SAMv2)
        iv. PIL (Pillow)
        v. NumPy
    d. Usage

```
python sam2_surgical_mask.py `
  --input images/surgery_scene.jpg `
  --output masks/surgical_mask.png `
  --model-type vit_h `
  --checkpoint sam_v2.pth `
  --device cuda
```

    e. Arguments
        i. –input (str) - Path to the input image (required)
        ii. --output (str) - Path where the masked PNG output should be saved (required)
        iii. --model-type (str) - SAMv2 model type. Default: vit_h
        iv. –checkpoint (str) - Path to the model checkpoint. Default: sam_v2.pth
        v. –device (str) - Device to run the model on (cuda or cpu). Default: cuda
        vi. –method (str) - Method for selecting the best mask (largest supported). Default: largest
    f. Function Descriptions
        i. load_image(path)
            - Loads an image and converts it to an RGB NumPy array.
        ii. init_sam(model_type, checkpoint, device)
            - Initializes the SAM model and automatic mask generator using a specified type, checkpoint, and device.
        iii. generate_masks(mask_generator, image_np)

- Generates segmentation masks for the entire input image.

iv. select_instrument_mask(masks, method='largest')
- Selects the best mask representing the surgical instrument. Currently supports:
  a. 'largest': Selects the mask with the largest area.

v. crop_bottom_fraction(image_np, mask, top_fraction=0.33)
- Crops the image and mask to exclude the top top_fraction and keep the bottom portion (typically where instruments are located).

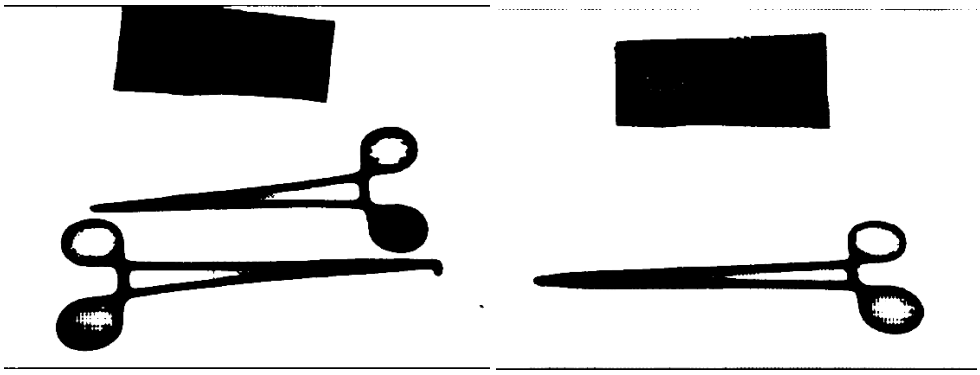
vi. export_masked_image(image_np, mask, out_path)
- Applies the selected mask to the cropped image, converting it to an RGBA format where the background is transparent. Saves the result as a PNG.

vii. segment_instrument(...)
- Executes the full segmentation pipeline described above.

g. Results

i. Keeping the Background's Color

ii. Black and White



iii. Keeping the Instrument's Color

7. **generate_masks_with_sam.py** – Binary Segmentation Masks
   a. Overview

   generate_masks_with_sam.py is a utility script designed to automatically generate binary segmentation masks from a labeled image dataset using SAM (Segment Anything Model) by Meta AI. It processes each image in a folder-structured dataset, applies object mask generation, and saves the resulting binary masks to disk in a mirrored folder structure. This script is ideal for creating pseudo-ground-truth segmentation labels from raw images, especially in weakly supervised or semi-automated annotation pipelines.

   b. Project Structure

```
Training Data Set/
├── ClassA/
│   ├── image1.jpg
│   └── image2.png
└── ClassB/
    └── image3.jpeg

generate_masks_with_sam.py
sam_vit_h_4b8939.pth

GeneratedMasks/        ← OUTPUT
├── ClassA/
│   └── image1.png   ← Binary mask
└── ClassB/
    └── image3.png
```

   c. Configuration
      i. Paths
         - input_dir = "Training Data Set"
            a. input_dir: Root folder of input images organized into subfolders by class.
         - output_dir = "GeneratedMasks"
            a. output_dir: Output directory where generated binary masks will be saved.

- sam_checkpoint = "sam_vit_h_4b8939.pth"
    a. sam_checkpoint: Path to the pre-trained SAM checkpoint file (must be downloaded from Meta's SAM GitHub repo).

ii. Model Type
- model_type = "vit_h"
- Choose from: "vit_h", "vit_l", or "vit_b" depending on your downloaded checkpoint.

iii. Supported Image Formats
- image_exts = ['.jpg', '.jpeg', '.png']

d. Model Setup

```
sam = sam_model_registry[model_type](checkpoint=sam_checkpoint)
sam.to(device=device)
mask_generator = SamAutomaticMaskGenerator(sam)
```

i. Loads the SAM model onto the appropriate device (GPU if available).
ii. Initializes the automatic mask generator with default parameters.

You may override SAM's default parameters (like IoU threshold or stability score) by passing them into SamAutomaticMaskGenerator(...).

e. Processing Pipeline
i. Iterate Over Input Folder
- Scans through subdirectories under input_dir, each representing a separate class label.

ii. Load Each Image
- Images are loaded using PIL and converted to RGB.
- Only images with extensions in image_exts are processed.

iii. Apply SAM
- The SAM model generates a list of object masks from the image.
- Each mask is a boolean segmentation array.

iv. Combine Masks
- All valid masks from SAM are combined into one binary mask using logical OR.
- This results in a single-channel binary mask image (object = white / 255, background = black / 0).

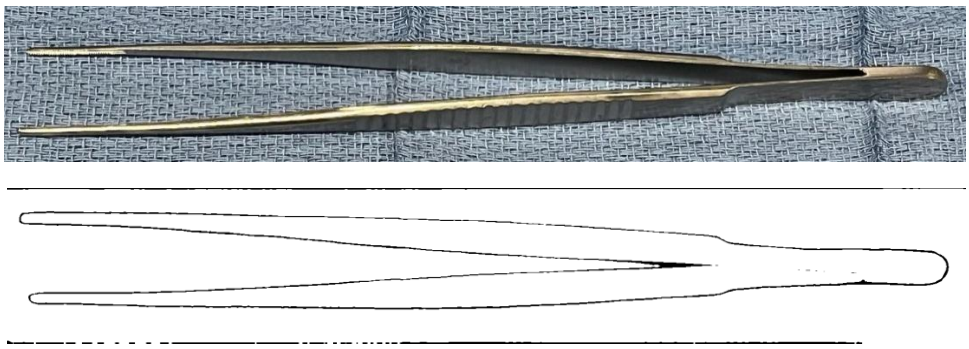
v. Save Output
- The final binary mask is saved to the corresponding class folder in output_dir.
- Saved in .png format using the original filename stem.

f. Example Usage
i. python generate_masks_with_sam.py
ii. Ensure the following:
- Your dataset is organized in class-labeled subfolders.
- The SAM checkpoint file exists at the path specified.

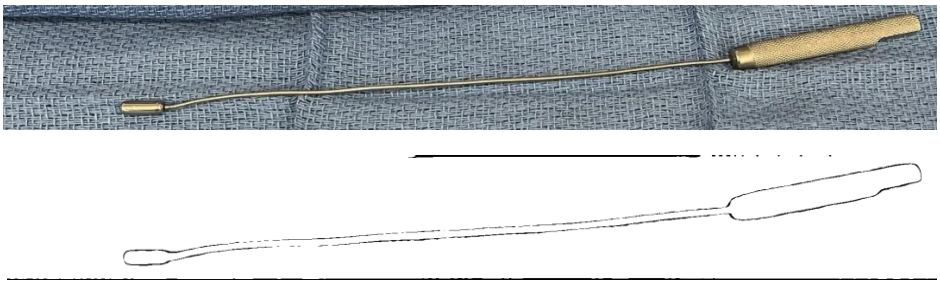
- You are in the same directory as the script (or adjust paths accordingly).

g. Notes
   i. No masks found: If SAM doesn't detect any objects, a warning is printed. You may need to adjust the input image resolution or fine-tune SAM's hyperparameters.
   ii. Customization: You can enhance mask quality by passing parameters like points_per_side, pred_iou_thresh, etc., into the SamAutomaticMaskGenerator constructor.
   iii. Performance: Processing is sequential and may be slow for large datasets. Consider adding multiprocessing if needed.

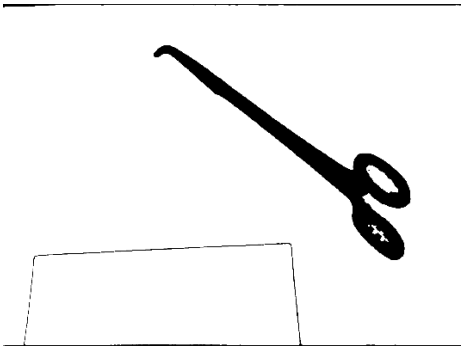
h. Results
   i. debakey_forceps_1.png



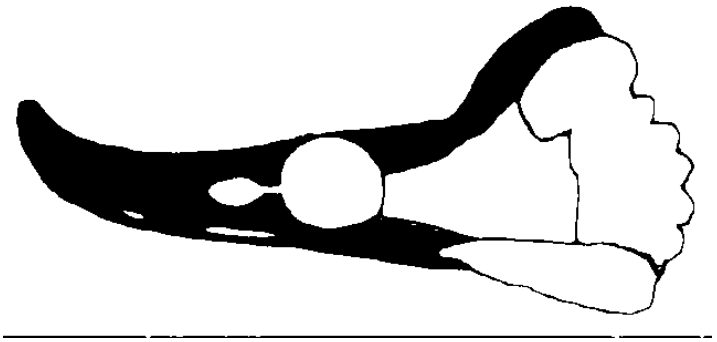
   ii. 1_4mm_dilators_1.png



   iii. right_angle_clamp_10.png



iv. angled_bulldog_clamp_large_1.png



8. **Model LoRA 1: inference.py** – Inference and Multi-Object Detection Using SAM2 + ViT+LoRA
    a. Purpose:
    Performs inference with the fine-tuned ViT model from model.py. Supports:
        i. Whole image classification (single object per image)
        ii. Multi-object detection and classification using SAM2 (Segment Anything Model v2)
    b. Main Functions:
        i. load_classifier(model_dir, base_model_name=None, device='cuda')
        Loads:
            - Fine-tuned ViT model with LoRA
            - Image processor (for preprocessing)
            - Label names (from label_names.txt or model config)
        ii. classify_patch(model, processor, image_patch, device='cuda')
            - Classifies a cropped patch of the image, returning the predicted label index and confidence.
        iii. simple_classify(image_path, model, processor, device='cuda')
            - Classifies the entire image and returns the predicted instrument class and confidence.
        iv. segment_and_count(...)
            - Uses SAM2 to:
                a. Segment multiple objects from an image
                b. Crop and classify each segment using the ViT+LoRA model

c. Return a dictionary of {label: count} and list of per-object detection metadata

c. CLI Arguments:
--model_dir - Directory with the fine-tuned ViT+LoRA model
--base_model - Optional base model name (if different from model_dir)
--image - Path to the input image
--use_sam - Flag to enable SAM-based segmentation
--sam_checkpoint - Path to a SAM2 checkpoint file

d. Output (console):
i. If --use_sam: Prints per-class counts of detected objects
ii. If not: Prints the top predicted class for the whole image

e. Requirements:
i. transformers, torch, peft, PIL, numpy, segment_anything (if using --use_sam)

f. Example Usage:
i. Training:

```
python model.py `
  --data_dir "./train" `
  --test_dir "./test" `
  --model_name "google/vit-base-patch16-224-in21k" `
  --output_dir "./vit-lora-finetuned" `
  --batch_size 16 `
  --epochs 5 `
  --lr 5e-5
```

ii. Inference (single object):

```
python inference.py `
  --model_dir "./vit-lora-finetuned" `
  --image "./sample.jpg"
```

iii. Inference (multi-object with SAM2):

```
python inference.py `
  --model_dir "./vit-lora-finetuned" `
  --image "./grouped_instruments.jpg" `
  --use_sam `
  --sam_checkpoint "./sam_vit_h.pth"
```

9. **Model LoRA 1: model.py** – Fine-Tuning ViT with LoRA for Instrument Classification
   a. Purpose:
   This script fine-tunes a Vision Transformer (ViT) model with LoRA (Low-Rank Adaptation) on a labeled image dataset of musical or surgical instruments (or any other classes). It optionally evaluates the model on a test set and saves the trained model for later use.
   b. Main Features:
      i. Loads and preprocesses image datasets (--data_dir and --test_dir).

ii. Applies LoRA adaptation to a pretrained ViT model (e.g., google/vit-base-patch16-224-in21k).
iii. Trains using standard cross-entropy loss and AdamW optimizer.
iv. Evaluates using metrics: accuracy, precision, recall, F1 score, log-loss, ROC-AUC.
v. Provides an opportunity to apply the LoRA technique for fast fine-tuning the model as new data comes in.
vi. Outputs:
- Fine-tuned model checkpoint
- label_names.txt for class decoding

c. Key Arguments:
--data_dir - Directory of training data (structured as subfolders per class)
--test_dir - Optional directory for testing
--model_name - Base ViT model name from Hugging Face
--output_dir - Directory to save fine-tuned model
--batch_size - Batch size for training/evaluation
--epochs - Number of training epochs
--lr - Learning rate for optimizer

d. Outputs:
i. A directory with:
- Fine-tuned ViT+LoRA model weights
- label_names.txt with one label per line
- Training logs printed to console

10. **Model LoRA 2: inference.py** – Semantic Segmentation
a. Purpose
Performs semantic segmentation on a single test image using a fine-tuned SegFormer model to detect and count surgical instruments.
b. Key Components
i. Configuration Section: Defines paths for the input image and model, and sets the target image size.
ii. Model Loading:
- Loads a pretrained SegFormer model from Hugging Face (nvidia/segformer-b0-finetuned-ade-512-512).
- Applies a LoRA adapter (via peft) for fine-tuned performance on surgical tools.
iii. Image Processing:
- Opens and resizes the test image to match model input size.
- Converts the image to a tensor format for inference.
iv. Inference:
- Passes the image through the model to generate segmentation logits.
- Upsamples logits to match target resolution and generates a prediction mask.

v. Post-processing:
- Counts predicted pixels per instrument class (excluding background).
- Displays a segmented mask overlay using matplotlib.

c. Usage Notes
i. Make sure to update image_path with the path to your test image.
ii. The model must be pre-trained and saved in the specified model_path.

11. **Model LoRA 2: surgical_instrument_segmentation.py** – Semantic Segmentation Model (SegFormer with LoRA)
a. Purpose
Trains a semantic segmentation model (SegFormer with LoRA) to detect surgical instruments in images, using a custom dataset structured by instrument classes.
b. Key Components
i. Configuration Section: Sets paths, model checkpoint, hyperparameters, and training parameters.
ii. Dataset Definition (InstrumentDataset):
- Loads images (and optionally masks) from class-based subdirectories.
- Supports synthetic masks if real segmentation masks are not provided.
iii. Transforms:
- transform_train: Applies resizing, flipping, rotation, and color augmentations.
- transform_val: Applies resizing for evaluation without augmentation.
iv. Data Loading:
- Splits dataset into training and validation subsets.
- Uses PyTorch DataLoader for batching and shuffling.
v. Model Setup:
- Loads a base SegFormer model with Hugging Face Transformers.
- Applies LoRA-based parameter-efficient fine-tuning via peft.
- Prints trainable parameter stats.
vi. Training Loop:
- Uses CrossEntropyLoss for segmentation task.
- Evaluates model on validation set after each epoch.
- Applies learning rate scheduling with warmup.
c. Optional Extensions
i. Provides an opportunity to apply the LoRA technique for fast fine-tuning the model as new data comes in.
ii. Add real segmentation masks for better supervision.
iii. Save training logs or validation predictions for analysis.
iv. Use mask_root if ground truth segmentation masks exist.

12. **Model YOLO LoRA** – YOLOv8 Custom Training Pipeline
    a. Overview
    This Python script implements a full pipeline for training a custom object detection model using YOLOv8. It takes a folder of raw class-labeled images, converts them into YOLO format, trains the model, and performs inference. It includes evaluation metrics to assess the performance of the model.
    b. Structure
        i. Configuration
            RAW_DATASET_DIR = "Training Data Set"
            YOLO_DATASET_DIR = "yolo_dataset"
            DATA_YAML_PATH = "custom_data.yaml"
            EPOCHS = 20
            IMG_SIZE = 640
            - Specifies paths for raw and YOLO-formatted data.
            - Sets training hyperparameters (number of epochs and image size).
        ii. Function: convert_to_yolo_format()
            - Converts the raw dataset (organized by class folders) into YOLO-compatible format.
            - Simulates bounding boxes for classification (bounding box covers the entire image).
            - Outputs:
                a. /images/train/*.jpg: Training images
                b. /labels/train/*.txt: Corresponding YOLO label files
                c. custom_data.yaml: Configuration file required by YOLOv8
            - Returns a class_map dictionary for label-integer mappings.
        iii. Function: train_model()
            - Loads a pre-trained YOLOv8 model (yolov8n.pt) and fine-tunes it on the custom dataset.
            - Performs training on CPU (can be switched to GPU with 'cuda').
            - Saves results to runs/detect/train.
        iv. Function: run_inference(model_path, test_image, class_map)
            - Runs inference on a single test image.
            - Loads the trained model and predicts object classes in the image.
            - Outputs a dictionary of detected class labels and their counts.
    c. Evaluation Metrics Used
    After training, the following evaluation metrics are automatically generated and logged by Ultralytics YOLOv8:
        i. Confusion Matrix: Displays prediction accuracy across all classes.
        ii. Normalized Confusion Matrix: Shows relative misclassification rates.
        iii. F1 Curve: Plots the F1 score across different confidence thresholds.
        iv. Labels Distribution Plot: Visualizes class imbalance in dataset.
        v. Labels Correlogram: Displays co-occurrence and correlation of labels.
        vi. P Curve (Precision): Shows precision at varying confidence thresholds.

vii. R Curve (Recall): Shows recall over different confidence levels.
viii. PR Curve (Precision-Recall): Joint view of precision and recall tradeoffs.
ix. Train Batches Metrics: Tracks performance over training batches.
x. Val Batches Metrics: Tracks performance over validation batches.
xi. Results are documented: Training logs, plots, and weights are saved in the runs/detect/train/ directory.

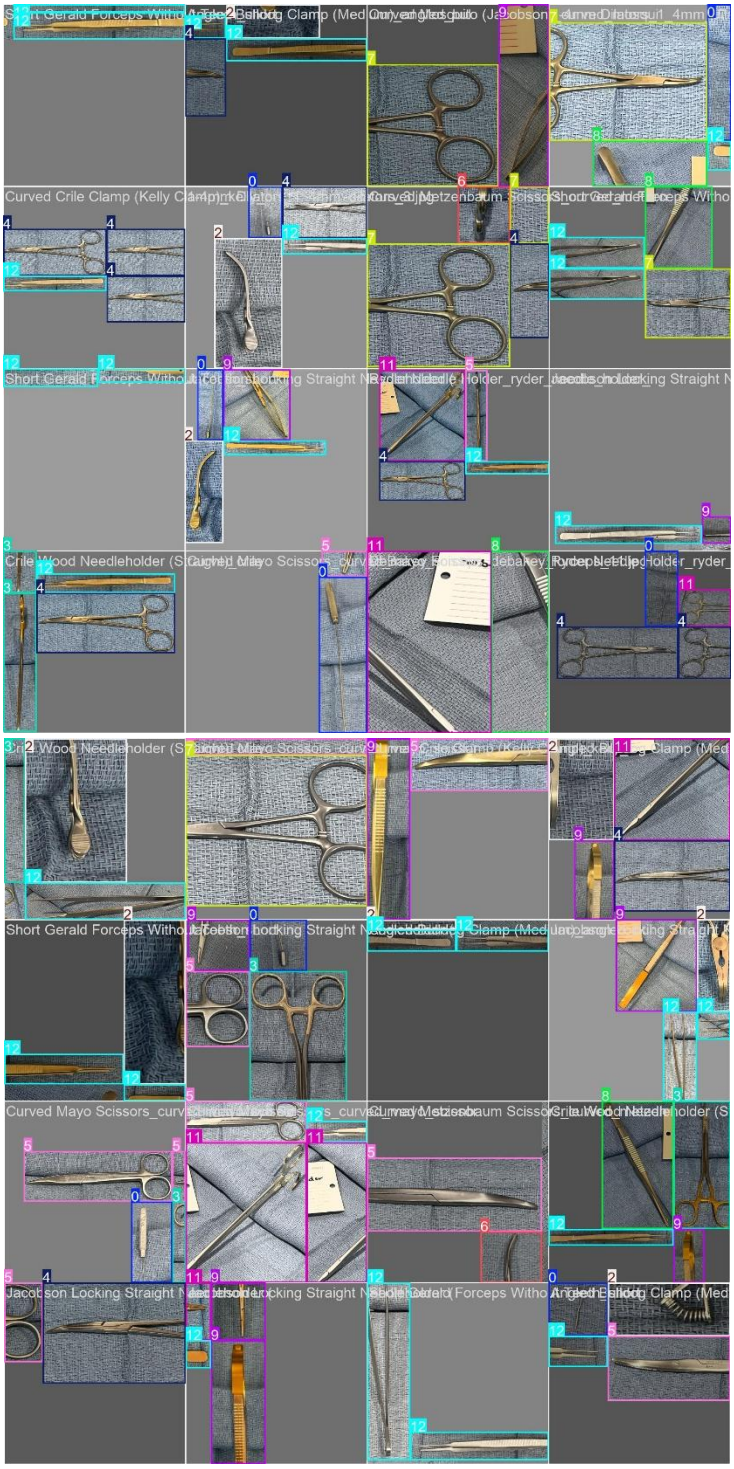
d. How to Run
   i. Prepare your dataset:
      - Create a directory with one subfolder per class, each containing relevant images.
   ii. Run the script
   iii. Run optional inference:
      - Provide a test image and use run_inference() to see detection results.

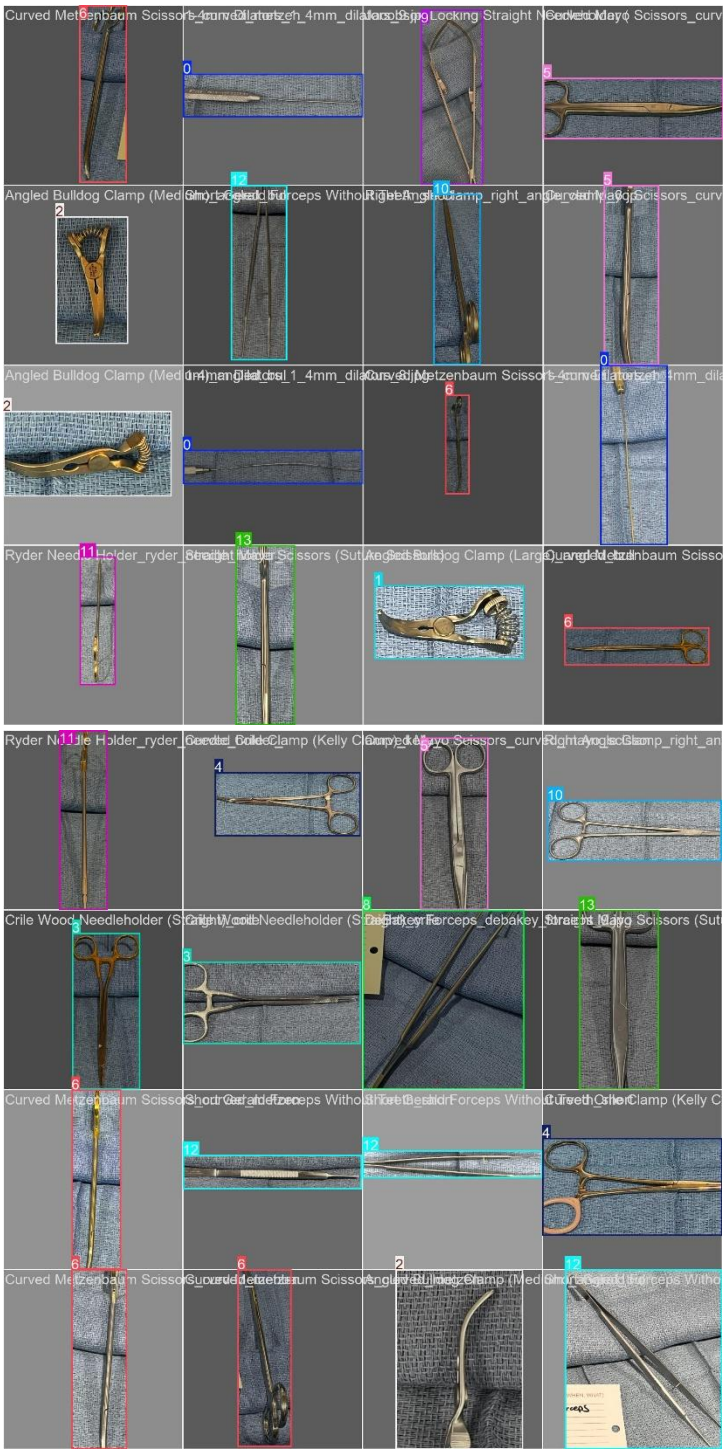
e. Output Directory Structure

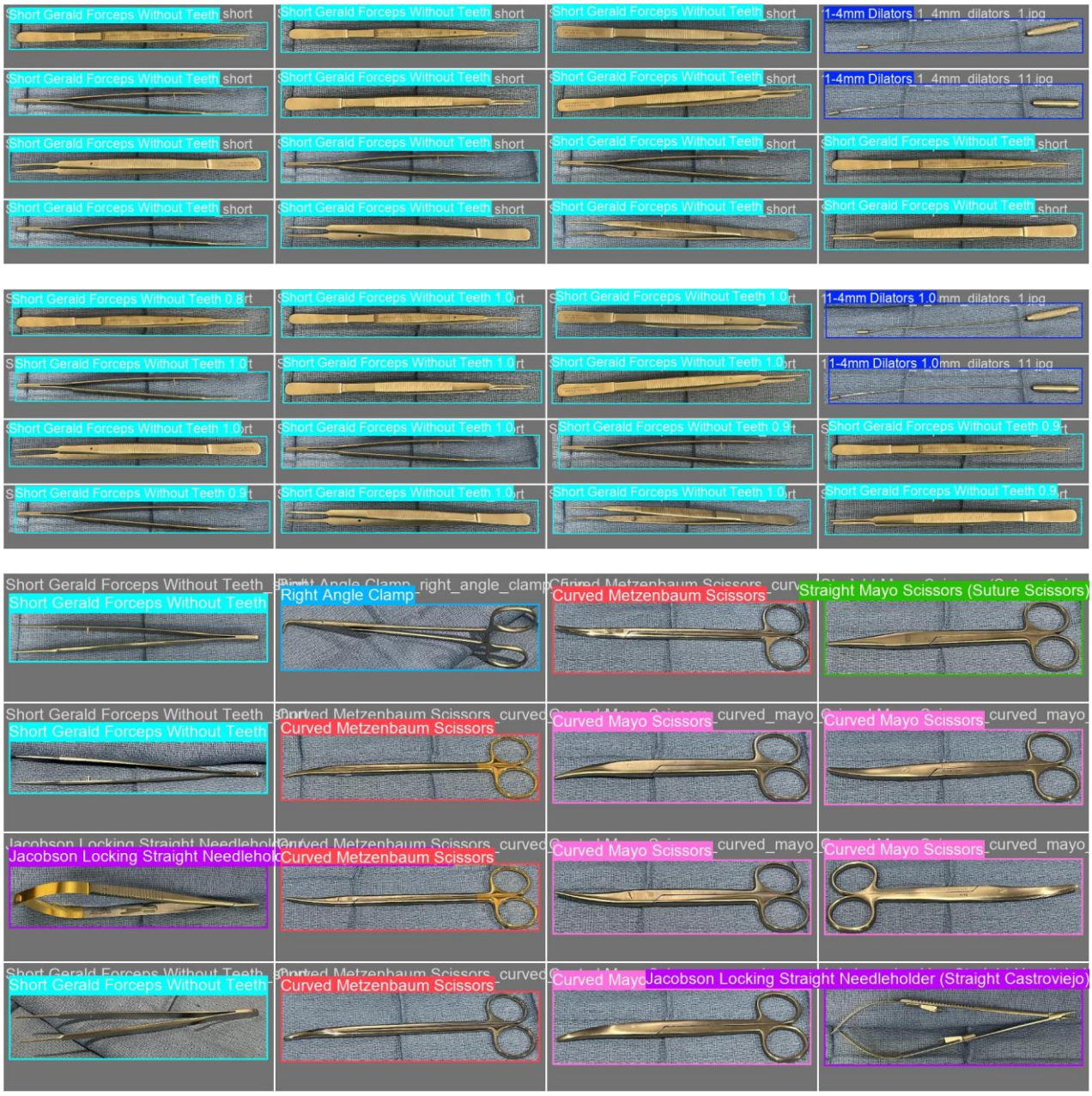
```
yolo_dataset/
│
├── images/
│   └── train/
│       ├── classA_img1.jpg
│       ├── ...
│
├── labels/
│   └── train/
│       ├── classA_img1.txt
│       ├── ...
│
└── custom_data.yaml
```

f. Notes
   i. Provides an opportunity to apply the LoRA technique for fast fine-tuning the model as new data comes in. YOLO could be used for fine-tuning too.
   ii. This script assumes image classification with bounding boxes covering the entire image.
   iii. Suitable for datasets where each image contains only one object centered and fully visible.
   iv. For multi-object detection, real bounding box annotations are required.

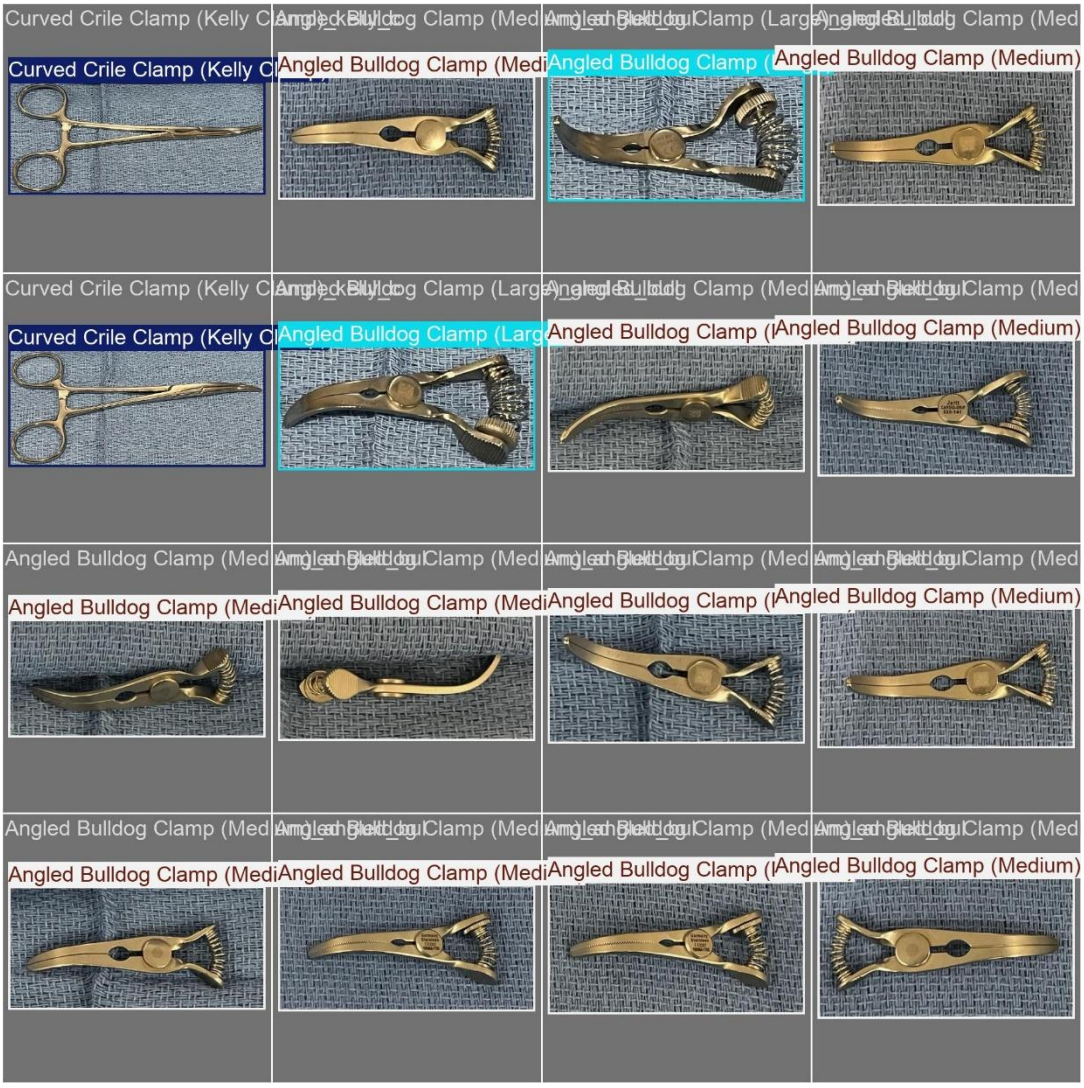
g. Results

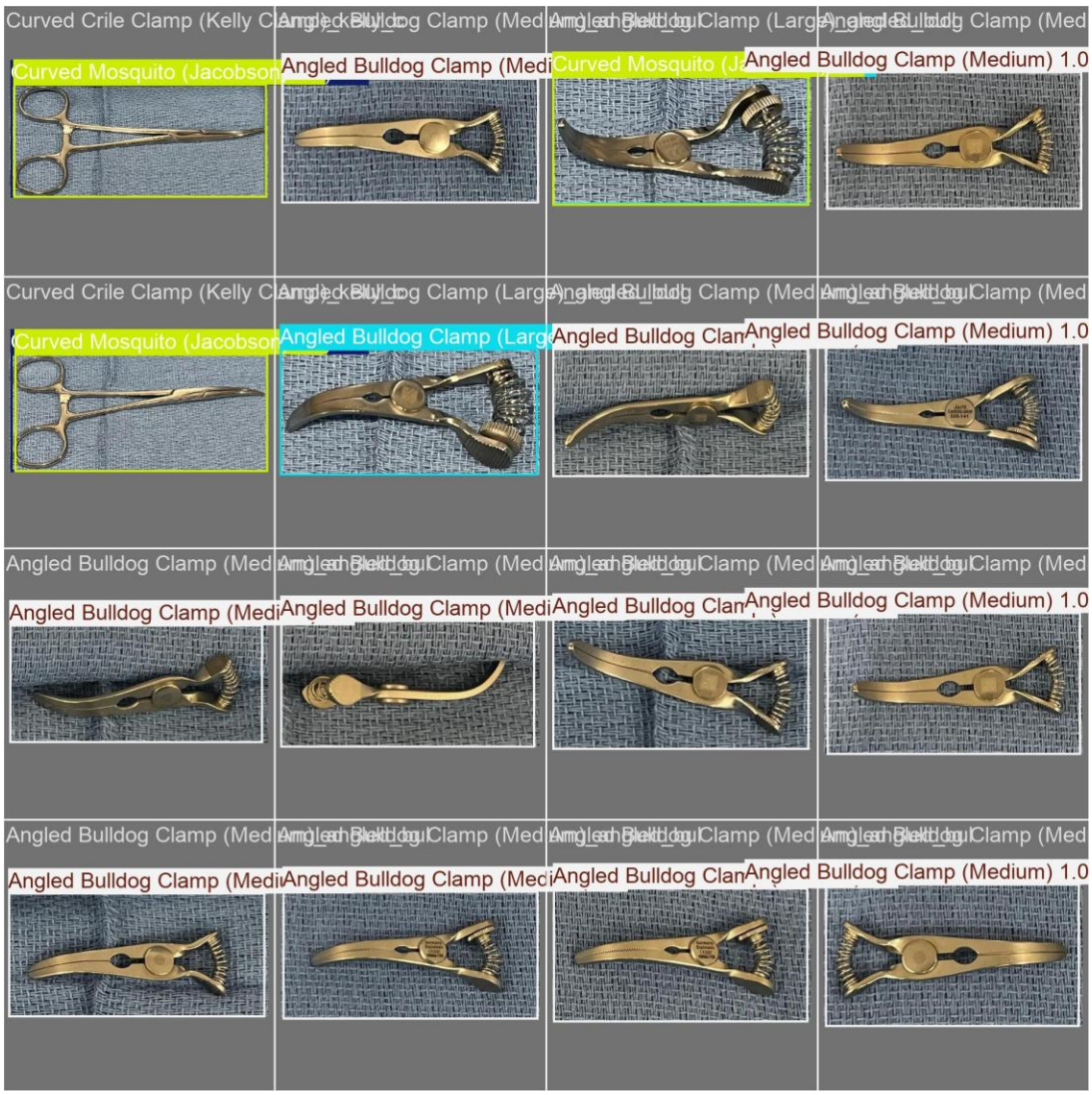


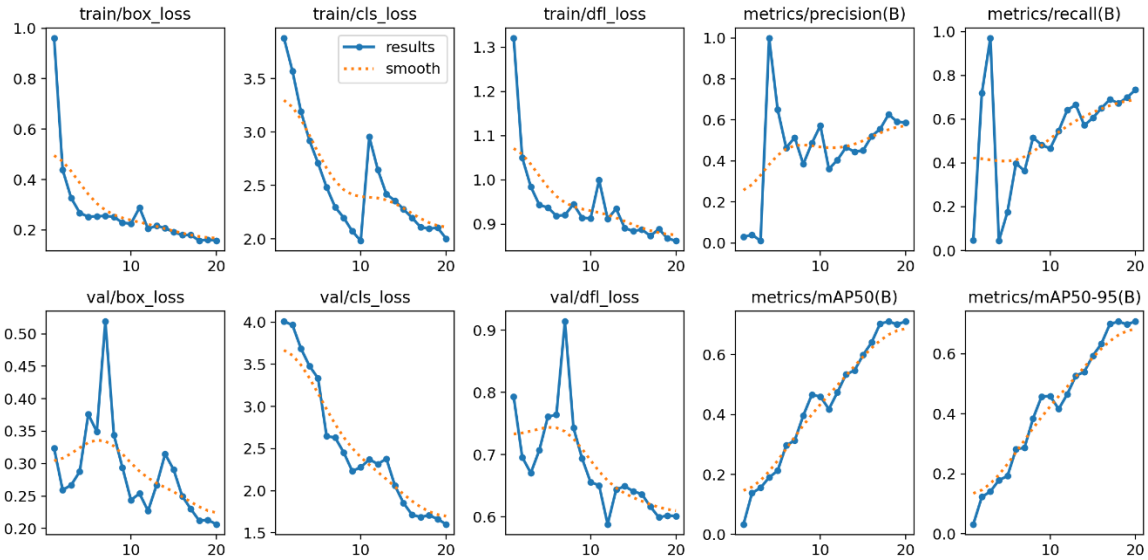


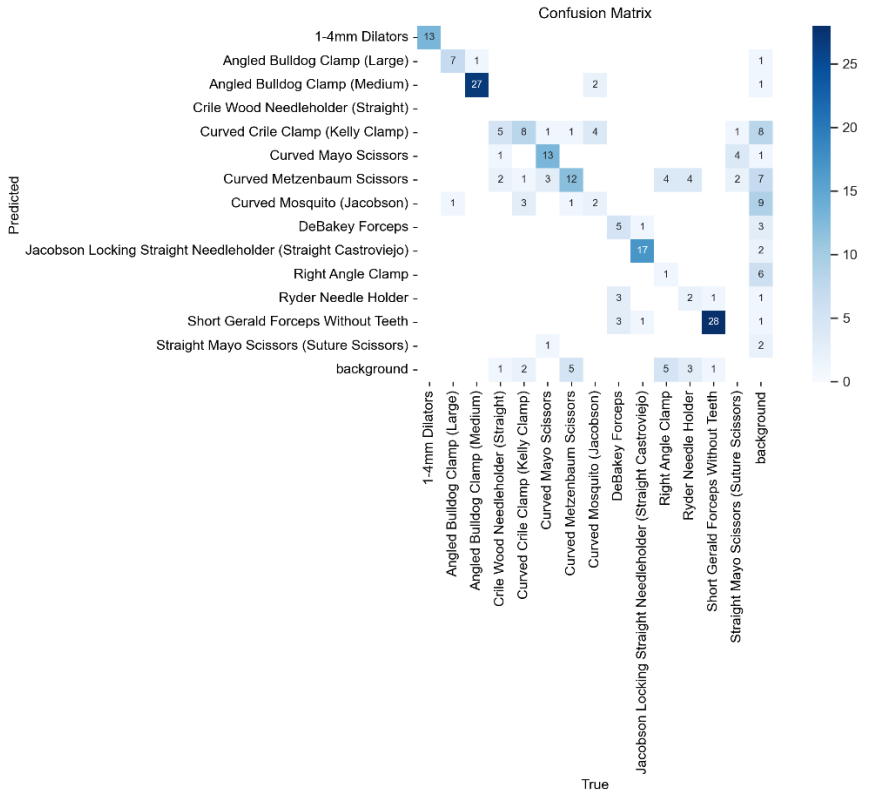

Short Gerald Forceps Without Teeth
1-4mm Dilators
Short Gerald Forceps Without Teeth 0.8
Short Gerald Forceps Without Teeth 1.0
Short Gerald Forceps Without Teeth 0.9
Right Angle Clamp
Curved Metzenbaum Scissors
Straight Mayo Scissors (Suture Scissors)
Curved Mayo Scissors
Jacobson Locking Straight Needleholder (Straight Castroviejo)

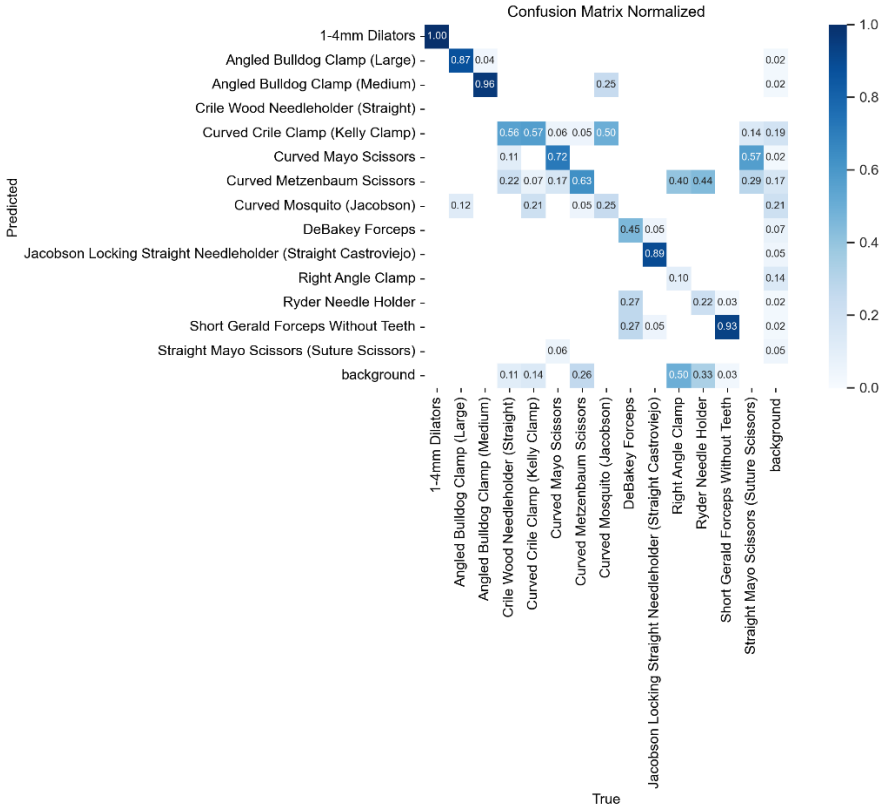

Short Gerald Forceps Without Teeth
Curved Metzenbaum Scissors 0.4
Curved Metzenbaum Scissors 0.3
Curved Mayo Scissors 0.3
Short Gerald Forceps Without Teeth 0.5
Curved Mayo Scissors 0.4
Curved Mayo Scissors 0.7
Short Gerald Forceps Without Teeth 0.6
(Straight Castroviejo) 0.7
Curved Mayo Scissors 0.5
Curved Mayo Scissors 0.6
Curved Metzenbaum Scissors 0.3
Jacobson Locking Straight Needleholder (Straight Castroviejo) 0.7

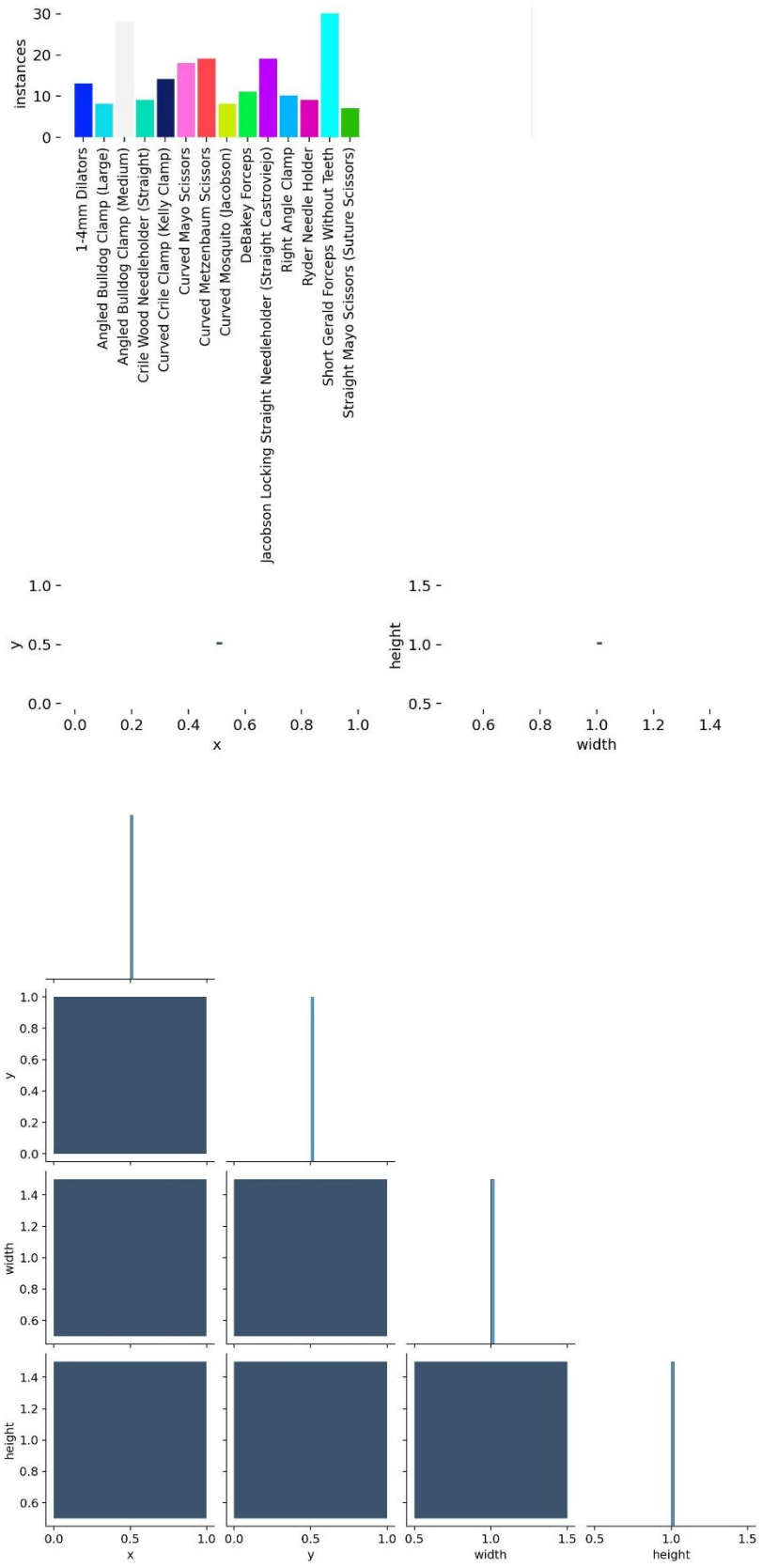

Curved Crile Clamp (Kelly C
Angled Bulldog Clamp (Medium)

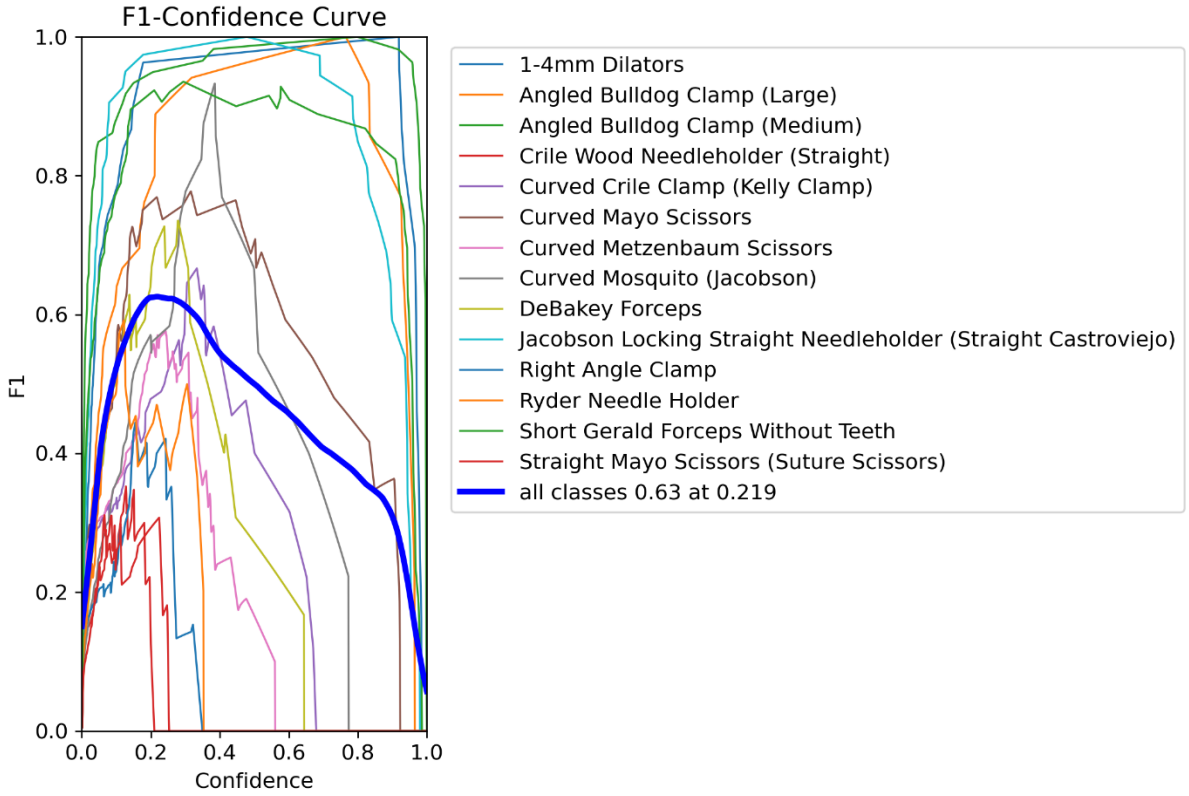

Curved Mosquito (Jacobson
Angled Bulldog Clamp (Medium) 1.0

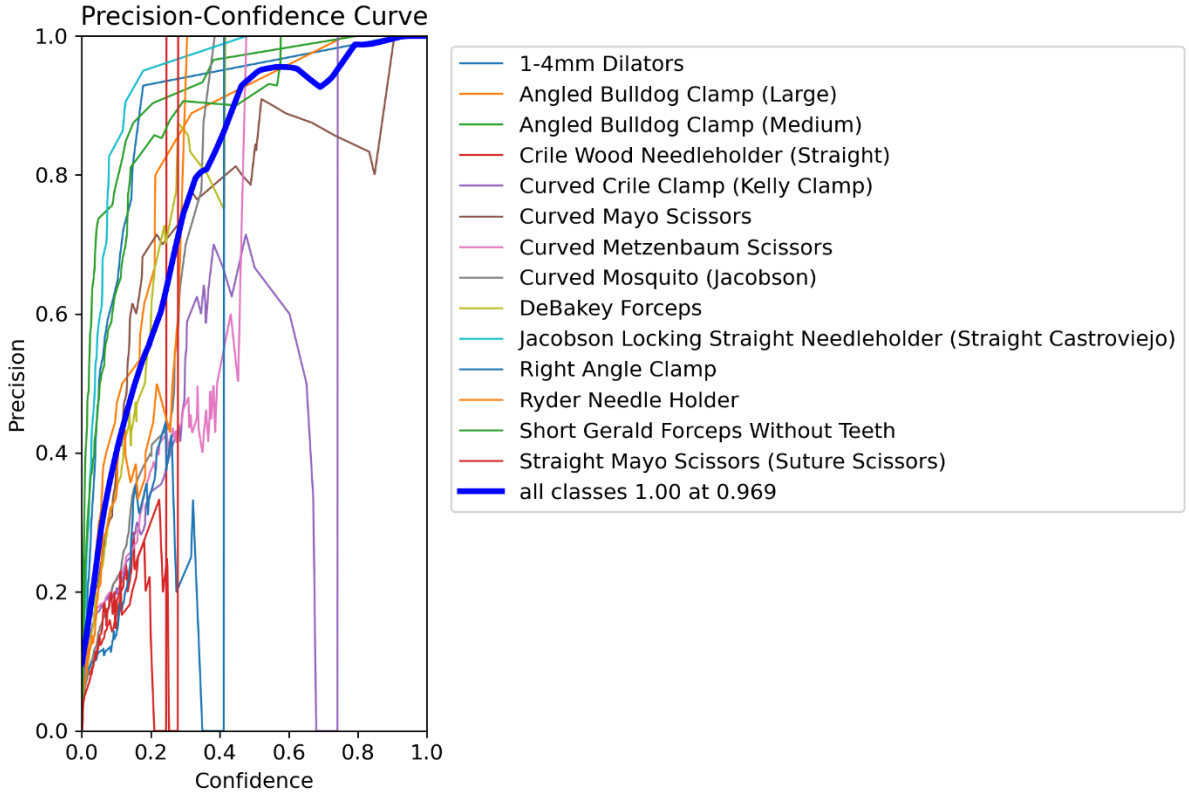

train/box_loss
train/cls_loss
results
smooth
train/dfl_loss
metrics/precision(B)
metrics/recall(B)
val/box_loss
val/cls_loss
val/dfl_loss
metrics/mAP50(B)
metrics/mAP50-95(B)

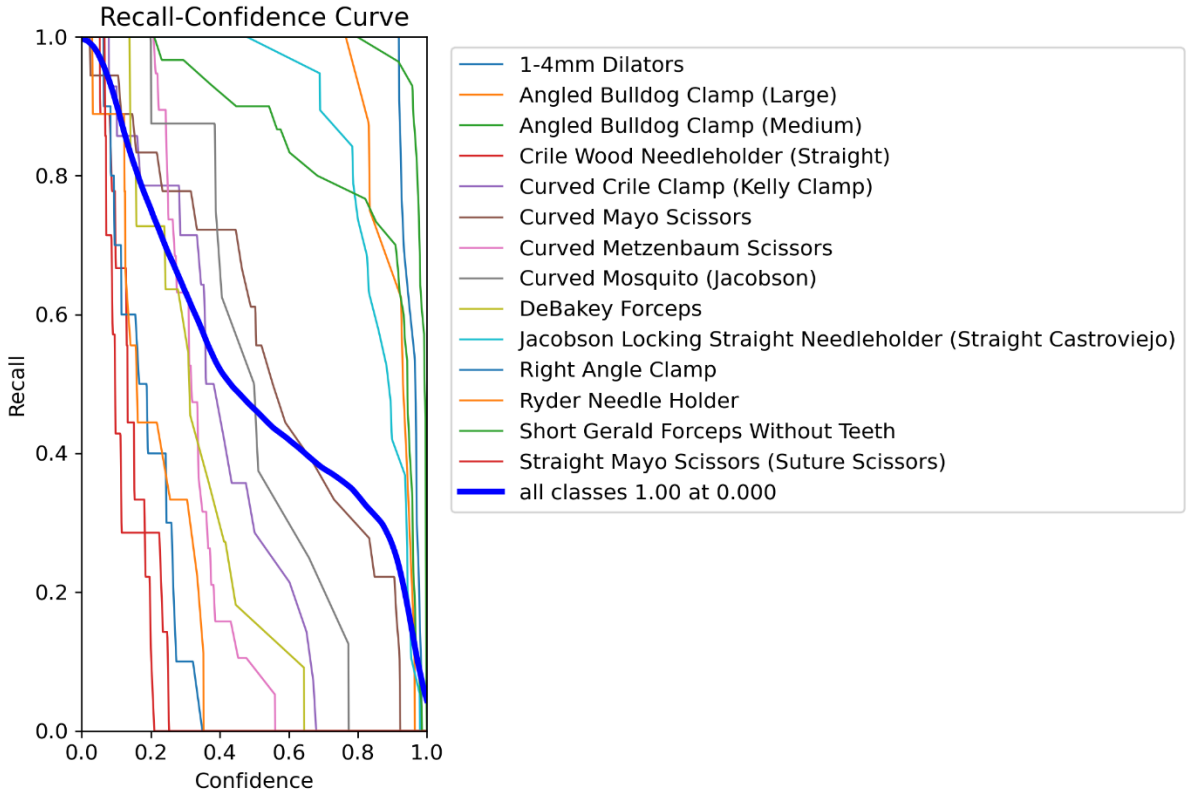

Confusion Matrix
Predicted
True
1-4mm Dilators
Angled Bulldog Clamp (Large)
Angled Bulldog Clamp (Medium)
Crile Wood Needleholder (Straight)
Curved Crile Clamp (Kelly Clamp)
Curved Mayo Scissors
Curved Metzenbaum Scissors
Curved Mosquito (Jacobson)
DeBakey Forceps
Jacobson Locking Straight Needleholder (Straight Castroviejo)
Right Angle Clamp
Ryder Needle Holder
Short Gerald Forceps Without Teeth
Straight Mayo Scissors (Suture Scissors)
background

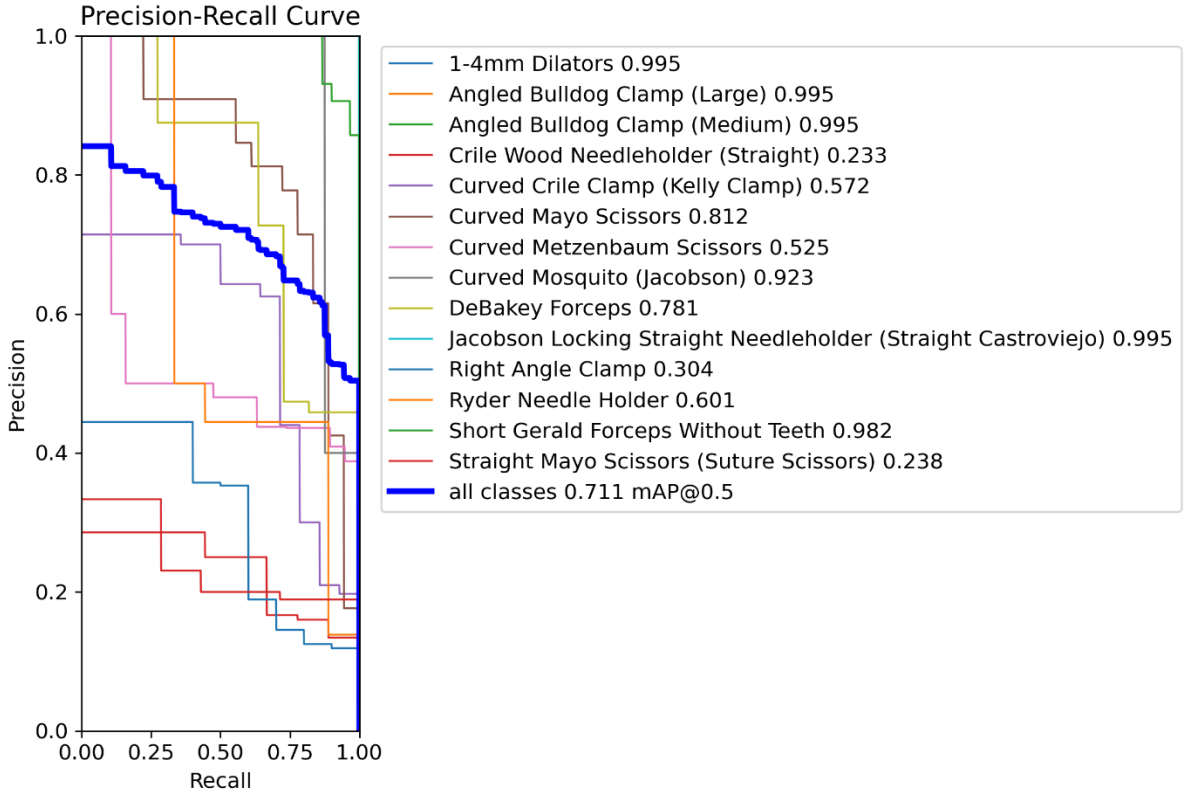

Confusion Matrix Normalized
Predicted
True
1-4mm Dilators
Angled Bulldog Clamp (Large)
Angled Bulldog Clamp (Medium)
Crile Wood Needleholder (Straight)
Curved Crile Clamp (Kelly Clamp)
Curved Mayo Scissors
Curved Metzenbaum Scissors
Curved Mosquito (Jacobson)
DeBakey Forceps
Jacobson Locking Straight Needleholder (Straight Castroviejo)
Right Angle Clamp
Ryder Needle Holder
Short Gerald Forceps Without Teeth
Straight Mayo Scissors (Suture Scissors)
background


instances
1-4mm Dilators
Angled Bulldog Clamp (Large)
Angled Bulldog Clamp (Medium)
Crile Wood Needleholder (Straight)
Curved Crile Clamp (Kelly Clamp)
Curved Mayo Scissors
Curved Metzenbaum Scissors
Curved Mosquito (Jacobson)
DeBakey Forceps
Jacobson Locking Straight Needleholder (Straight Castroviejo)
Right Angle Clamp
Ryder Needle Holder
Short Gerald Forceps Without Teeth
Straight Mayo Scissors (Suture Scissors)
x
y
width
height


F1-Confidence Curve
F1
Confidence
0.0
0.2
0.4
0.6
0.8
1.0
1-4mm Dilators
Angled Bulldog Clamp (Large)
Angled Bulldog Clamp (Medium)
Crile Wood Needleholder (Straight)
Curved Crile Clamp (Kelly Clamp)
Curved Mayo Scissors
Curved Metzenbaum Scissors
Curved Mosquito (Jacobson)
DeBakey Forceps
Jacobson Locking Straight Needleholder (Straight Castroviejo)
Right Angle Clamp
Ryder Needle Holder
Short Gerald Forceps Without Teeth
Straight Mayo Scissors (Suture Scissors)
all classes 0.63 at 0.219


Precision-Confidence Curve
Precision
Confidence
0.0
0.2
0.4
0.6
0.8
1.0
1-4mm Dilators
Angled Bulldog Clamp (Large)
Angled Bulldog Clamp (Medium)
Crile Wood Needleholder (Straight)
Curved Crile Clamp (Kelly Clamp)
Curved Mayo Scissors
Curved Metzenbaum Scissors
Curved Mosquito (Jacobson)
DeBakey Forceps
Jacobson Locking Straight Needleholder (Straight Castroviejo)
Right Angle Clamp
Ryder Needle Holder
Short Gerald Forceps Without Teeth
Straight Mayo Scissors (Suture Scissors)
all classes 1.00 at 0.969


Recall-Confidence Curve
Recall
Confidence
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.2 0.4 0.6 0.8 1.0
1-4mm Dilators
Angled Bulldog Clamp (Large)
Angled Bulldog Clamp (Medium)
Crile Wood Needleholder (Straight)
Curved Crile Clamp (Kelly Clamp)
Curved Mayo Scissors
Curved Metzenbaum Scissors
Curved Mosquito (Jacobson)
DeBakey Forceps
Jacobson Locking Straight Needleholder (Straight Castroviejo)
Right Angle Clamp
Ryder Needle Holder
Short Gerald Forceps Without Teeth
Straight Mayo Scissors (Suture Scissors)
all classes 1.00 at 0.000


Precision-Recall Curve
Precision
Recall
1.0
0.8
0.6
0.4
0.2
0.0
0.00 0.25 0.50 0.75 1.00
1-4mm Dilators 0.995
Angled Bulldog Clamp (Large) 0.995
Angled Bulldog Clamp (Medium) 0.995
Crile Wood Needleholder (Straight) 0.233
Curved Crile Clamp (Kelly Clamp) 0.572
Curved Mayo Scissors 0.812
Curved Metzenbaum Scissors 0.525
Curved Mosquito (Jacobson) 0.923
DeBakey Forceps 0.781
Jacobson Locking Straight Needleholder (Straight Castroviejo) 0.995
Right Angle Clamp 0.304
Ryder Needle Holder 0.601
Short Gerald Forceps Without Teeth 0.982
Straight Mayo Scissors (Suture Scissors) 0.238
all classes 0.711 mAP@0.5