# Assignment 3
# Report

Computer Vision

Samridh Girdhar

2021282

# Question 1

## 1 CLIP Installation and Setup

Following the OpenAI CLIP GitHub repository instructions, we install the required dependencies and load the model:

```
# Install required dependencies
!conda install --yes -c pytorch pytorch=1.7.1 torchvision cudatoolkit=11.0
!pip install ftfy regex tqdm
!pip install git+https://github.com/openai/CLIP.git
```

## 2 Loading CLIP

```
import torch
import clip
from PIL import Image

# Load the CLIP model with pretrained weights (ViT-B/32)
device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load("ViT-B/32", device=device)
print(f"Model loaded on {device}")
```

## 3 CLIP Analysis of Sample Image

Using the loaded CLIP model, we analyze a sample image of a person holding a dog:

```
# Load and preprocess the image
image = Image.open("sample_image.jpg")
image_input =
↪  preprocess(image).unsqueeze(0).to(device)

# textual descriptions for comparison
captions = [
    "A man holding a large dog",
    "A gray Great Dane being held by its owner",
    "A person with a small puppy",
    "A dog sitting on a couch",
    "A man in formal attire with a pet",
    "A woman holding a cat",
    "A gray horse in a stable",
    "A person standing next to a bookshelf",
    "A large dog with its owner in a living
    ↪  room",
    "A man in a white shirt holding a gray dog"
]

# Tokenize the text descriptions
text_inputs = torch.cat([clip.tokenize(c) for c
↪  in captions]).to(device)

# Calculate features and similarities
with torch.no_grad():
    image_features =
    ↪  model.encode_image(image_input)
    text_features =
    ↪  model.encode_text(text_inputs)

    # Normalize features
    image_features = image_features /
    ↪  image_features.norm(dim=-1, keepdim=True)
    text_features = text_features /
    ↪  text_features.norm(dim=-1, keepdim=True)

    # Calculate similarity scores
    similarity = (100.0 * image_features @
    ↪  text_features.T).softmax(dim=-1)

    # Print the results
    print("\nCLIP Similarity Scores:")
    for i, caption in enumerate(captions):
        print(f"{caption}:
        ↪  {similarity[0][i].item():.2%}")
```

Figure 1: Sample image used for caption similarity

### CLIP Results

```
CLIP Similarity Scores:
A man holding a large dog: 5.95%
A gray Great Dane being held by its owner: 78.37%
A person with a small puppy: 0.14%
A dog sitting on a couch: 0.01%
A man in formal attire with a pet: 1.21%
A woman holding a cat: 0.02%
A gray horse in a stable: 0.02%
A person standing next to a bookshelf: 0.00%
A large dog with its owner in a living room: 0.63%
A man in a white shirt holding a gray dog: 13.62%
```

## 4 CLIPS Installation

Following the instructions from the CLIPS GitHub repository:

```
# Install required dependencies
# pip install -r requirements.txt from CLIPS repo
# pip install open-clip-torch
```

## 5 CLIPS loading

```
import torch
import torch.nn.functional as F
from PIL import Image
from open_clip import create_model_from_pretrained, get_tokenizer

# Load the CLIPS-Large-14-224 model
model, preprocess =
↪  create_model_from_pretrained('hf-hub:UCSC-VLAA/ViT-L-14-CLIPS-Recap-DataComp-1B')
tokenizer = get_tokenizer('hf-hub:UCSC-VLAA/ViT-L-14-CLIPS-Recap-DataComp-1B')
```

## 6 CLIPS Analysis of Sample Image

Using the loaded CLIPS model, we analyze the same image with the same captions:

```
# Load and preprocess the image
image = Image.open("dog_human.jpg")
```

```
image_input = preprocess(image).unsqueeze(0)

# Use the same captions from the CLIP example
# Process with CLIPS
with torch.no_grad(), torch.cuda.amp.autocast() if torch.cuda.is_available() else
↪  torch.no_grad():
    # Tokenize all captions
    text_tokens = tokenizer(captions, context_length=model.context_length)

    # Encode image and text
    image_features = model.encode_image(image_input)
    text_features = model.encode_text(text_tokens)

    # Normalize features
    image_features = F.normalize(image_features, dim=-1)
    text_features = F.normalize(text_features, dim=-1)

    # Calculate similarity scores
    similarity = (100.0 * image_features @ text_features.T).softmax(dim=-1)

    # Print the results
    print("\nCLIPS Similarity Scores:")
    for i, caption in enumerate(captions):
        print(f"{caption}: {similarity[0][i].item():.2%}")
```

### CLIPS Results

```
CLIPS Similarity Scores:
A man holding a large dog: 76.11%
A gray Great Dane being held by its owner: 16.04%
A person with a small puppy: 0.01%
A dog sitting on a couch: 0.00%
A man in formal attire with a pet: 1.57%
A woman holding a cat: 0.00%
A gray horse in a stable: 0.00%
A person standing next to a bookshelf: 0.03%
A large dog with its owner in a living room: 0.77%
A man in a white shirt holding a gray dog: 5.46%
```

## 7 Analysis of Results

The comparison between CLIP and CLIPS reveals several interesting insights:

### Key Differences

- **Different Top Predictions**:
  - CLIP strongly favors *"A gray Great Dane being held by its owner"* (78.37%)
  - CLIPS strongly favors *"A man holding a large dog"* (76.11%)
- **Specificity vs. Generality**:
  - CLIP gives higher scores to specific descriptions (breed identification)
  - CLIPS gives higher scores to more general but accurate descriptions
- **Secondary Preferences**:
  - CLIP's second choice is *"A man in a white shirt holding a gray dog"* (13.62%)
  - CLIPS's second choice is *"A gray Great Dane being held by its owner"* (16.04%)

### Shared Characteristics

Both models correctly assign very low probabilities to obviously incorrect descriptions like *"A woman holding a cat"* and *"A gray horse in a stable"*.

### Model Behavior Analysis

**CLIP** appears to be more confident in specific visual details like breed identification (Great Dane) and color attributes. This suggests CLIP may be better at fine-grained visual categorization when trained on web-crawled pairs that often contain specific nomenclature.

**CLIPS** seems to prioritize the overall scene description and relationship between subjects ("man holding large dog") over specific breed identification. This could reflect its training on synthetic captions that might focus more on relationships and actions rather than specific taxonomic labels.

### Practical Implications

These differences highlight how model training approaches affect what visual-textual relationships are emphasized:

1. **Use Case Considerations:**

- CLIP might be preferable for applications requiring fine-grained categorization or specific attribute recognition

- CLIPS might be better for applications focused on understanding relationships and actions between subjects

2. **Caption Style Preferences:**

- CLIP appears to favor descriptive, taxonomically precise captions

- CLIPS appears to favor action-oriented, relationship-focused captions

3. **Error Patterns:**

- Both models effectively reject completely incorrect descriptions

- Both models assign low probabilities to background elements that are present but not the main focus

## Question 2

## 1. BLIP Setup and Model

We used the paper **BLIP: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation** (arXiv:2201.12086) and the official GitHub repository: github.com/salesforce/BLIP.

The following dependencies were installed:

```
pip install torch torchvision torchaudio
pip install transformers timm pillow
```

We used the HuggingFace version of the BLIP VQA model:

```
from transformers import BlipProcessor, BlipForQuestionAnswering
import torch
from PIL import Image

device = "cuda" if torch.cuda.is_available() else "cpu"
processor = BlipProcessor.from_pretrained("Salesforce/blip-vqa-base"
    ↪ )
model = BlipForQuestionAnswering.from_pretrained("Salesforce/blip-
    ↪ vqa-base").to(device)
```

## 2. Where is the dog present in the image?

**Input Image:**

**Code for Inference:**

```
image = Image.open("sample_image.jpg").convert("RGB")
question = "Where␣is␣the␣dog␣present␣in␣the␣image?"
```

```
inputs = processor(image, question, return_tensors="pt").to(device)
out = model.generate(**inputs)
answer = processor.decode(out[0], skip_special_tokens=True)
print("Answer:", answer)
```

**Output:**

```
Answer: in man's arms
```

## 3. Where is the man present in the image?

**Code for Inference:**

```
question = "Where␣is␣the␣man␣present␣in␣the␣image?"
inputs = processor(image, question, return_tensors="pt").to(device)
out = model.generate(**inputs)
answer = processor.decode(out[0], skip_special_tokens=True)
print("Answer:", answer)
```

**Output:**

```
Answer: living room
```

## 4. Comments on Output Accuracy

The answers generated by BLIP were contextually correct and semantically meaningful:

- The dog is clearly visible being carried in the man's arms, making `"in man's arms"` an accurate spatial reference, no finer localisation needed.
- The environment shows typical indoor features (bookshelf, wood floor, closed door), validating the caption `"living room"` as an appropriate room-level classification.

## Question 3

## 1. BLIP Image Captioning Model Setup

We used the **Salesforce BLIP (Bootstrapping Language-Image Pretraining)** model to generate captions from input images. The pretrained model weights were loaded from HuggingFace.

**Installation:**

```
pip install torch torchvision torchaudio
pip install transformers timm pillow
```

**Loading the model:**

```
from transformers import BlipProcessor, BlipForConditionalGeneration
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"
processor = BlipProcessor.from_pretrained("Salesforce/blip-image-
    ↪ captioning-base")
model = BlipForConditionalGeneration.from_pretrained(
    "Salesforce/blip-image-captioning-base"
).to(device).eval()
```

## 2. Generating Captions with BLIP

The sample images were extracted from the provided ZIP archive and passed to the BLIP model. For each image:

```
from PIL import Image

image = Image.open("example.jpg").convert("RGB")
inputs = processor(image, return_tensors="pt").to(device)
output = model.generate(**inputs)
caption = processor.decode(output[0], skip_special_tokens=True)
print("Caption:", caption)
```


(a) a small dog walking on a green carpet

(b) a small dog running across a green field

(c) a family sitting in a pool with a towel

(d) a small bird sitting on a plant

(e) a small dog standing on a stone ledge

(f) a man riding a bike down a wet street

(g) a brown butterfly sitting on a green plant

(h) a man in a suit and tie sitting on a couch

(i) a duck drinking water from a pond

(j) a coffee machine with two cups on it

Figure 2: Image captions generated by the model

## 3. Evaluating Semantic Accuracy using CLIP

We used the OpenAI CLIP model to evaluate the alignment between the image and the generated caption via cosine similarity.

**CLIP inference:**

```
from transformers import CLIPProcessor, CLIPModel

clip_model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32
   ↪ ").to(device)
clip_processor = CLIPProcessor.from_pretrained("openai/clip-vit-base
   ↪ -patch32")

inputs = clip_processor(text=[caption], images=image, return_tensors
   ↪ ="pt").to(device)
with torch.no_grad():
    outputs = clip_model(**inputs)

image_emb = outputs.image_embeds / outputs.image_embeds.norm(dim=-1,
   ↪  keepdim=True)
text_emb = outputs.text_embeds / outputs.text_embeds.norm(dim=-1,
```

```
    ↪ keepdim=True)
cosine_similarity = (image_emb * text_emb).sum().item()
```

**Results compiled in Table 0**

## 4. CLIPS (CLIPScore) Evaluation

CLIPS or CLIPScore is computed as cosine similarity × 100:

```
clip_score = cosine_similarity * 100
print("CLIPScore:", clip_score)
```

| Image | Caption | Cosine | Score |
|---|---|---|---|
| ILSVRC2012_test_00000003.jpg | a small dog walking on a green carpet | 0.316 | 31.57 |
| ILSVRC2012_test_00000004.jpg | a small dog running across a green field | 0.327 | 32.71 |
| ILSVRC2012_test_00000018.jpg | a family sitting in a pool with a towel | 0.313 | 31.34 |
| ILSVRC2012_test_00000019.jpg | a small bird sitting on a plant | 0.289 | 28.94 |
| ILSVRC2012_test_00000022.jpg | a small dog standing on a stone ledge | 0.310 | 31.04 |
| ILSVRC2012_test_00000023.jpg | a man riding a bike down a wet street | 0.324 | 32.39 |
| ILSVRC2012_test_00000025.jpg | a brown butterfly sitting on a green plant | 0.320 | 31.96 |
| ILSVRC2012_test_00000026.jpg | a man in a suit and tie sitting on a couch | 0.293 | 29.29 |
| ILSVRC2012_test_00000030.jpg | a duck drinking water from a pond | 0.312 | 31.17 |
| ILSVRC2012_test_00000034.jpg | a coffee machine with two cups on it | 0.291 | 29.10 |

Table 0: Results for Clip similarity and Score

## 5. Metrics for Caption–Image Alignment

Various metrics can be used to quantify the alignment between images and their generated captions:

- **Cosine Similarity (CLIP)**: Measures direct vector alignment. Best used in open-domain, zero-shot evaluations.
- **CLIPS / CLIPScore**: A scaled version of cosine, interpretable and widely used for caption quality.
- **CIDEr**: TF-IDF n-gram overlap with ground-truth captions. Ideal for dataset benchmarks (e.g., COCO).

- **SPICE**: Measures object/attribute/relationship match via scene graphs. Useful in evaluating detailed semantics.
- **BLEU / ROUGE / METEOR**: Older n-gram metrics, fast to compute, but less aligned with human perception.
- **Human Evaluation**: Ultimately necessary for critical applications (medical, legal) or subjective domains.

**Example Use Cases:**

- Use **CLIPScore** for evaluating caption quality in web-scale applications.
- Use **CIDEr** and **SPICE** when multiple ground-truth captions are available.
- Use **Cosine Similarity** when comparing embeddings across captioning models or ranking captions.

**When to Use What**

- **Exploratory or zero-shot settings** (no reference captions):
  *CLIP cosine / CLIPScore* are handy—immediate, reference-free, and correlate reasonably with human assessments.
- **Model development on COCO-style datasets**:
  Pair *CLIPScore* with *CIDEr* or *SPICE*, so you capture both semantic alignment and lexical diversity.
- **Application-specific tuning** (e.g., product search captions):
  Perform *retrieval recall*—does the caption uniquely find its image among similar items?
- **Deployment-critical outputs** (medical, legal):
  Always add a round of *human evaluation*, even if automated scores look high.

| Metric | What it measures | Good for | Caveats |
|---|---|---|---|
| **Cosine similarity (raw CLIP)** | Angular distance between CLIP image & text embeddings | Quick sanity-check of semantic match; ranking captions for one image | Uncalibrated; values vary with model/layer; not directly comparable across setups |
| **CLIPScore / CLIPS** | Cosine × 100 (sometimes length-penalized) | Reporting caption quality with a single number; correlates well with human judgment | Still inherits CLIP bias; higher isn't always better for specificity vs. generality |
| **CIDEr** | n-gram TF-IDF similarity against multiple references | Traditional caption benchmarks (COCO, Flickr); rewards consensus wording | Needs ground-truth reference captions—unavailable for web images or zero-shot tasks |
| **SPICE** | Scene-graph overlap (objects, attributes, relations) | Evaluating semantic correctness beyond surface wording | Slower; depends on reliable scene-graph parsing; again needs reference captions |
| **BLEU / ROUGE / METEOR** | n-gram overlap | Historical baselines; cheap to compute | Weak correlation with human judgment, especially for open-vocabulary captions |
| **Image–Text Retrieval Recall (R@k)** | Does the caption retrieve its own image among distractors? | Dataset-level evaluation of alignment models | Requires a large gallery; only yields set-level statistics, not per-caption scores |
| **Human evaluation** | Direct judgment of relevance, fluency, detail | Final QA, user-facing applications | Expensive and slow; subjective variance |

Table 1: Overview of commonly used metrics for evaluating image captions.

# Question 4

## 1 Setup and Pretrained Models

The code was based on the LAVT GitHub repository:
`https://github.com/yz93/LAVT-RIS`

We created a separate environment using:

```
conda create -n lavt python=3.7 -y
conda activate lavt
pip install -r requirements.txt
pip install opencv-python matplotlib gdown
```

We downloaded pretrained weights:

```
# Swin backbone
wget https://github.com/SwinTransformer/storage/releases/download/v1
   ↪ .0.0/swin_base_patch4_window12_384_22k.pth -O
   ↪ pretrained_weights/swin_base.pth

# LAVT model checkpoint
gdown https://drive.google.com/uc?id=1
   ↪ xFMEXr6AGU97Ypj1yr8oo00uObbeIQvJ -O checkpoints/
   ↪ lavt_one_refcoco.pth
```

## 2 Segmentation with Ground Truth References

For each sample image, we extracted the natural language description from `reference.txt` and used the LAVT model to perform segmentation.

### Code Snippet:

```
def run_one(img_path, text, tag):
    # 1) load + preprocess image
    img = Image.open(img_path).convert('RGB')
    w, h = img.size
    sample = {'image': img, 'text': text}
    sample = T(sample)
    sample = {k: v.unsqueeze(0).to(device) if torch.is_tensor(v)
       ↪ else v
              for k,v in sample.items()}
```

```
    # 2) tokenize
    tokens = tokenizer.encode_plus(
        text,
        add_special_tokens=True,
        max_length=32,
        padding='max_length',
        truncation=True,
        return_tensors='pt'
    )
    input_ids      = tokens['input_ids'].to(device)
    attention_mask = tokens['attention_mask'].to(device)

    # *** inspect signature if in doubt ***
    # import inspect; print(inspect.signature(model.forward))

    # 3) forward WITH mask
    with torch.no_grad():
        out = model(sample['image'], input_ids, attention_mask)

    #
       ↪
       ↪
    # 1) Extract the     pred_masks     tensor however  i t s  packaged
       ↪ :
    if isinstance(out, dict):
        masks = out.get('pred_masks', None)
        if masks is None:
            raise KeyError("`out` dict has no 'pred_masks' key")
    elif isinstance(out, (tuple, list)):
        masks = out[0]   # assume first entry is the mask
    else:
        masks = out      # model returned a bare tensor

    # 2) Inspect to be sure:
    print(f">>> mask container type: {type(masks)}, shape: {tuple(
       ↪ masks.shape)}")

    #
       ↪
       ↪
    # 3) Threshold to boolean
    masks = masks.sigmoid() > 0.5  # still e.g. [1,1,H,W] or [1,H,W]

    # 4) Peel off all leading dims until we hit a 2 D  mask:
    while masks.dim() > 2:
        masks = masks[0]
```

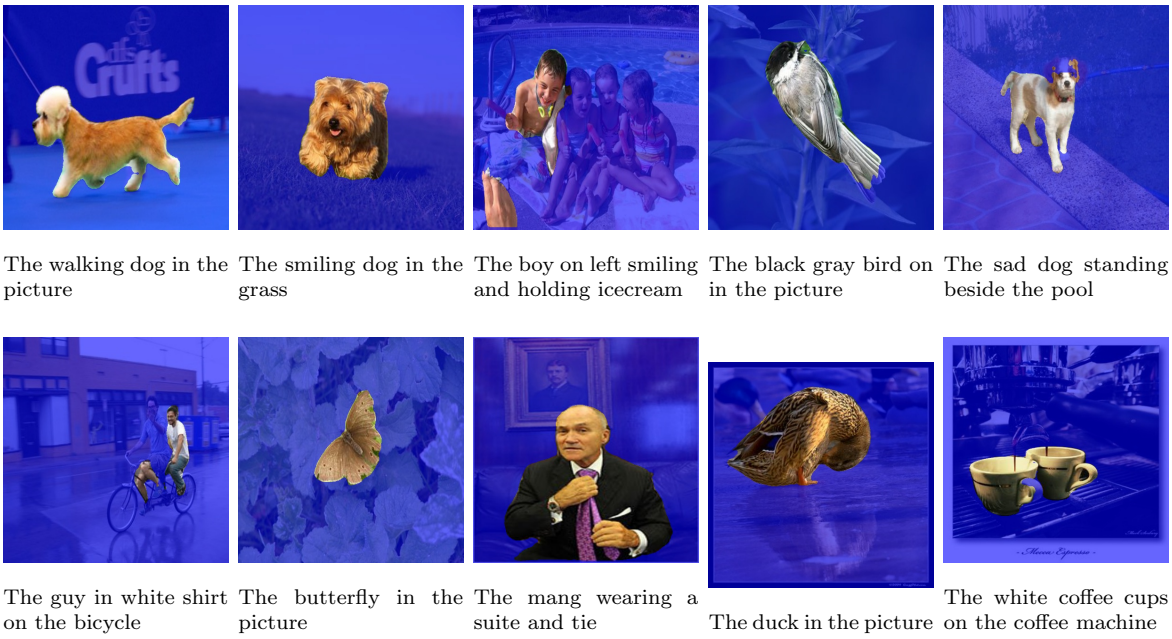
```
        mask = masks.cpu().numpy()  # shape (H_mask, W_mask)

    # 5) overlay on *resized* image
    img_cv = cv2.cvtColor(np.array(img), cv2.COLOR_RGB2BGR)
    img_cv = cv2.resize(img_cv, (mask.shape[1], mask.shape[0]))
    vis    = overlay_mask(img_cv, mask)
    cv2.imwrite(f"{OUT_DIR}/{tag}_overlay.jpg", vis)
    cv2.imwrite(f"{OUT_DIR}/{tag}_mask.png", (mask*255).astype('
       ↪ uint8'))

    heat = feature_heatmap(
    y1_feat['val'],                  # already [C,H,W]
    (mask.shape[0], mask.shape[1])   # target size
)
    cv2.imwrite(f"{OUT_DIR}/{tag}_y1.jpg",
            cv2.applyColorMap(heat, cv2.COLORMAP_JET))


for fname, ref in references.items():
    run_one(os.path.join(SAMPLES_DIR, fname), ref, fname.split('.')
       ↪ [0] + '_GT')
```

### Ground Truth Segmentation Results



The walking dog in the picture | The smiling dog in the grass | The boy on left smiling and holding icecream | The black gray bird on in the picture | The sad dog standing beside the pool

The guy in white shirt on the bicycle | The butterfly in the picture | The mang wearing a suite and tie | The duck in the picture | The white coffee cups on the coffee machine

Figure 3: Overlayed segmentation results using ground-truth reference text for all 10 sample images

## 3 Y1 Feature Map Visualization

Y1 feature maps (first stage Swin features) were extracted using a forward hook and visualized by averaging across channels.

### Code Snippet:

```
def y1_hook(_, __, output):
    if isinstance(output, (tuple, list)):
        x_out, H, W = output[0], output[1], output[2]
        feat = x_out[0].transpose(0, 1).reshape(-1, H, W)
    else:
        feat = output.detach()[0]
    y1_feat['val'] = feat.detach()
```

### Y1 Feature Maps

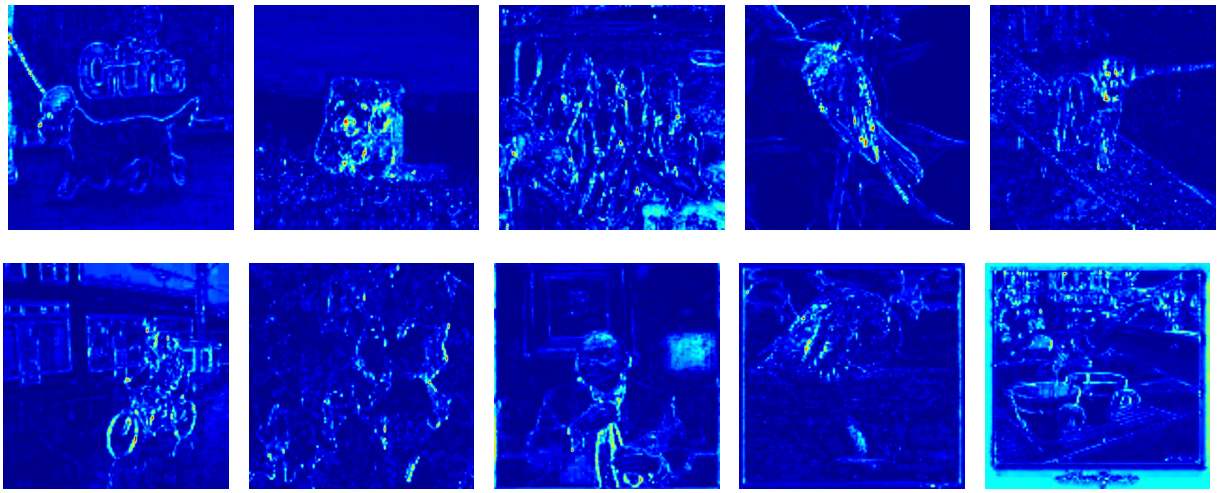


Figure 4: Y1 feature maps for all 10 ground-truth reference samples

## 4 Failure Segmentation with Custom References (2 points)

We crafted misleading reference texts to evaluate robustness.

### Custom Failure Prompts

- 00000003: *A cat sleeping on a sofa*

- 00000004: *A red car parked by the tree*
- 00000018: *A group of people playing football*
- 00000019: *A dog running in the park*
- 00000022: *A boat sailing on the ocean*
- 00000023: *A child painting on a canvas*
- 00000025: *A person riding a horse*
- 00000026: *A man cooking in a kitchen*
- 00000030: *A car driving down a highway*
- 00000034: *A row of bicycles lined up*

**Code Snippet:**

```
for fname, prompt in my_prompts.items():
    run_one(os.path.join(SAMPLES_DIR, fname), prompt, fname.split('.
       ↪ ')[0] + '_MYFAIL')
```

**Failure Segmentation Results**


Figure 5: Failure segmentation results using adversarial prompts for all 10 images

You can find detailed results here.

# Question 5

## 1 Installation and Setup

### Repository and Installation

We referred to the official Matcher GitHub repository:

`https://github.com/aim-uofa/Matcher`

Following the README instructions:

- Cloned the repository
- Installed dependencies including `detectron2`, `tensorboardX`, and `fvcore`
- Downloaded the pre-trained weights for DINOv2 and SAM:
  - `dinov2_vitl14_pretrain.pth`
  - `sam_vit_h_4b8939.pth`

### Code Snippet: Environment Setup

```
# clone repo
https://github.com/aim-uofa/Matcher
cd Matcher

# install dependencies
pip install -r requirements.txt

# download weights
wget https://dl.fbaipublicfiles.com/dinov2/dinov2_vitl14_pretrain.
   ↪ pth -P models
wget https://dl.fbaipublicfiles.com/segment_anything/
   ↪ sam_vit_h_4b8939.pth -P models
```

## 2 One-shot Segmentation Results

Each subfolder contains two images. For every subfolder, we perform segmentation twice:

- Take the first image as reference and the second as target
- Then, reverse the roles

**Code Snippet: Running Matcher on Image Folders**

```
import os, cv2, torch, glob
from matcher.Matcher import build_matcher_oss
from types import SimpleNamespace

# Setup matcher config
args = SimpleNamespace(
    device = torch.device("cuda" if torch.cuda.is_available() else "
       ↪ cpu"),
    dinov2_weights = "models/dinov2_vitl14_pretrain.pth",
    sam_weights = "models/sam_vit_h_4b8939.pth",
    dinov2_size = "vit_large",
    sam_size = "vit_h",
    use_semantic_sam = False,
    points_per_side = 64,
    pred_iou_thresh = 0.88,
    sel_stability_score_thresh = 0.90,
    stability_score_thresh = 0.95,
    iou_filter = 0.0,
    box_nms_thresh = 1.0,
    output_layer = 3,
    use_dense_mask = 0,
    multimask_output = 0,
    dense_multimask_output = 0,
    num_centers = 8,
    use_box = False,
    use_points_or_centers = True,
    sample_range = (1,6),
    max_sample_iterations = 64,
    alpha = 1.0,
    beta = 0.0,
    exp = 0.0,
    emd_filter = 0.0,
    purity_filter = 0.0,
    coverage_filter = 0.0,
    use_score_filter = False,
    deep_score_filter = 0.33,
    deep_score_norm_filter = 0.10,
    topk_scores_threshold = 0.0,
    num_merging_mask = 9,
)
matcher = build_matcher_oss(args)

# Loop through folders
ROOT_IMG_DIR = "Images"
OUTPUT_DIR = "outputs/matcher_vis"
os.makedirs(OUTPUT_DIR, exist_ok=True)
```

```
for folder in sorted(os.listdir(ROOT_IMG_DIR)):
    imgs = sorted(glob.glob(os.path.join(ROOT_IMG_DIR, folder, '*.*g
        ↪ ')))
    ref_img = cv2.imread(imgs[0])
    tgt_img = cv2.imread(imgs[1])
    # Perform segmentation using matcher here
    # Save visualisation overlays and masks
```

### Example Outputs from Matcher

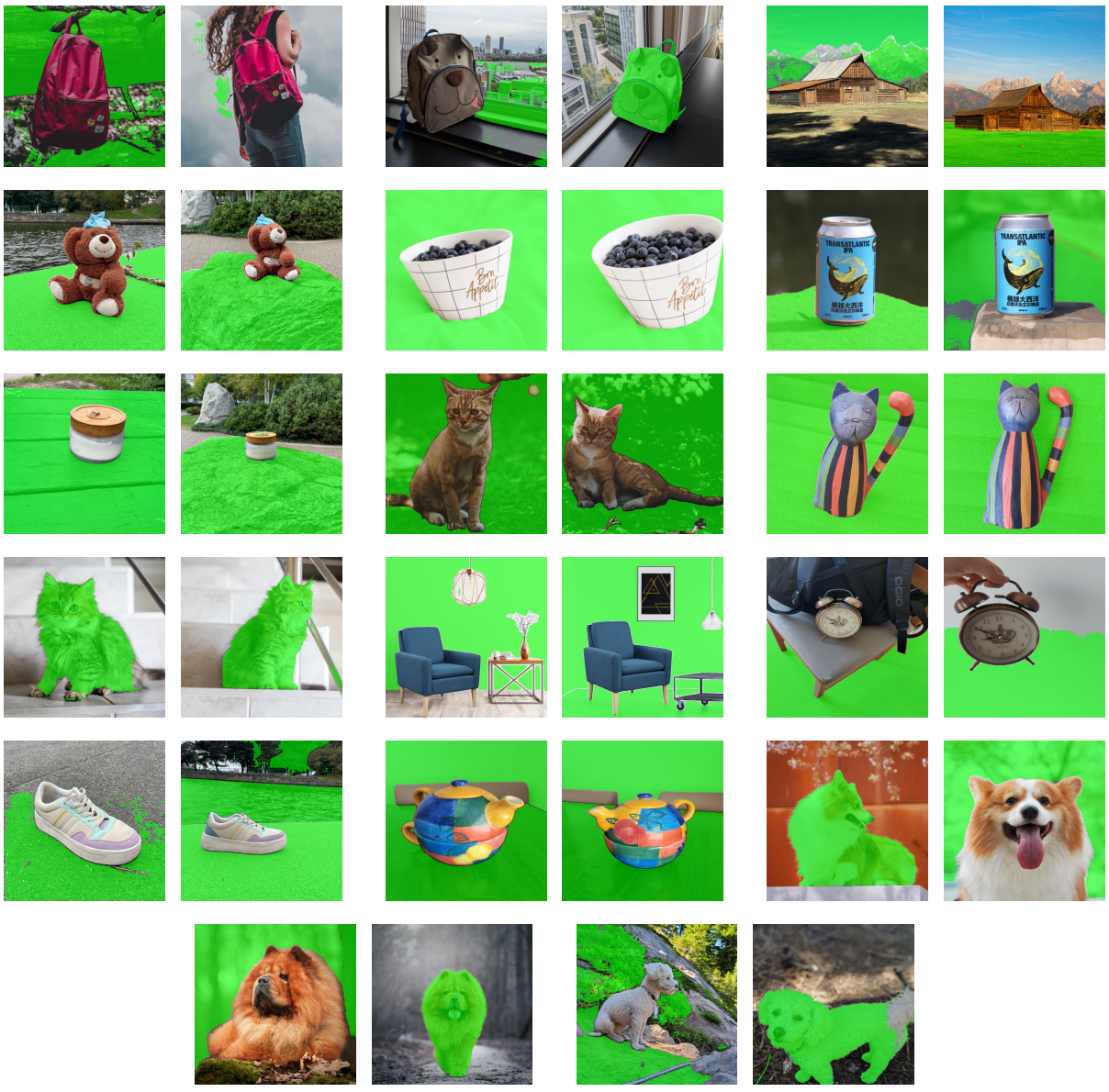


Figure 6: Matcher one-shot segmentation results (each folder: seg_1 and seg_2)

### Quantitative Output Files

Matcher also saves the predicted binary masks as `.npy` files:

- `mask_1_target.npy`
- `mask_2_reference.npy`

You can find detailed results here.