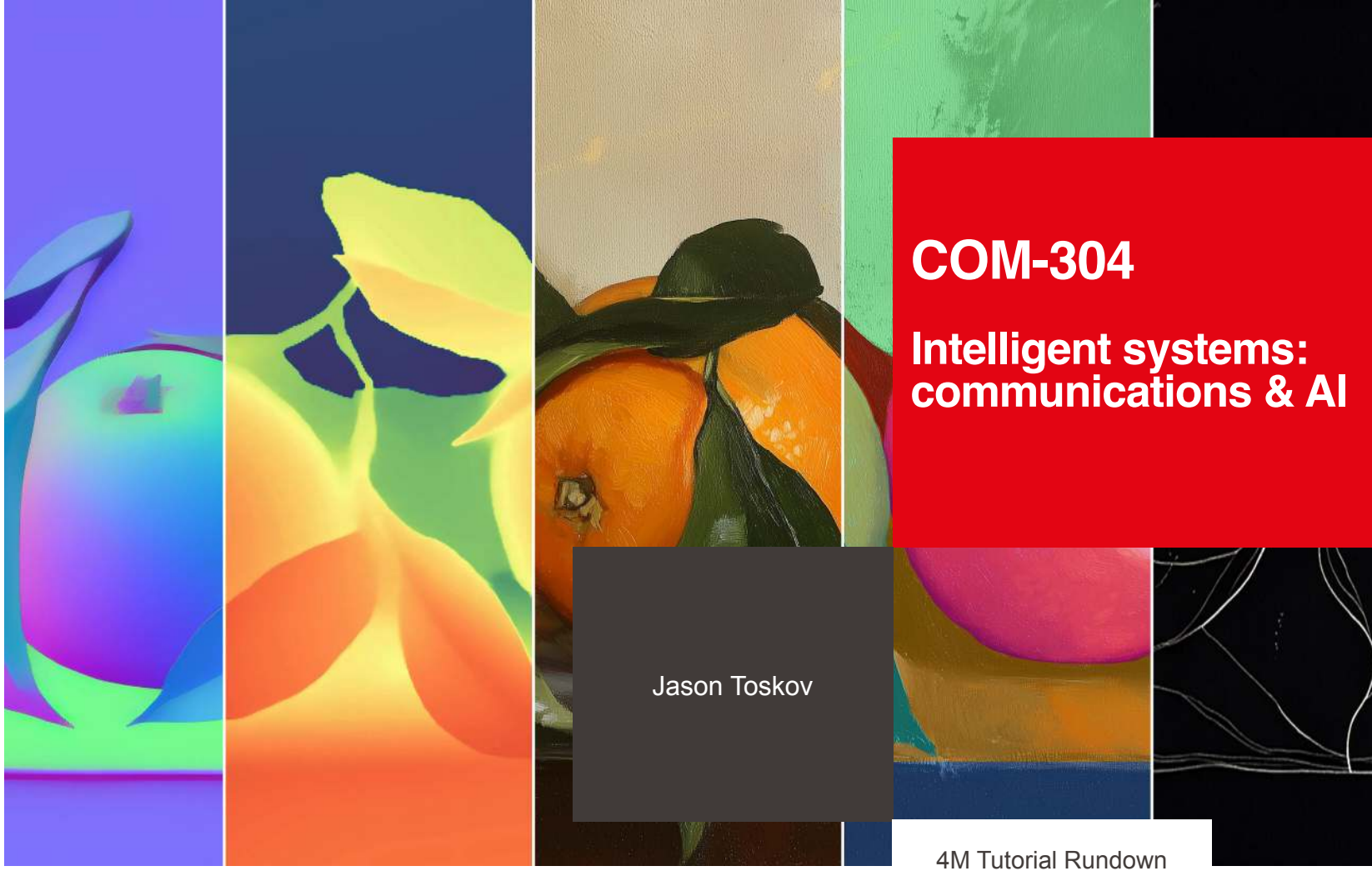


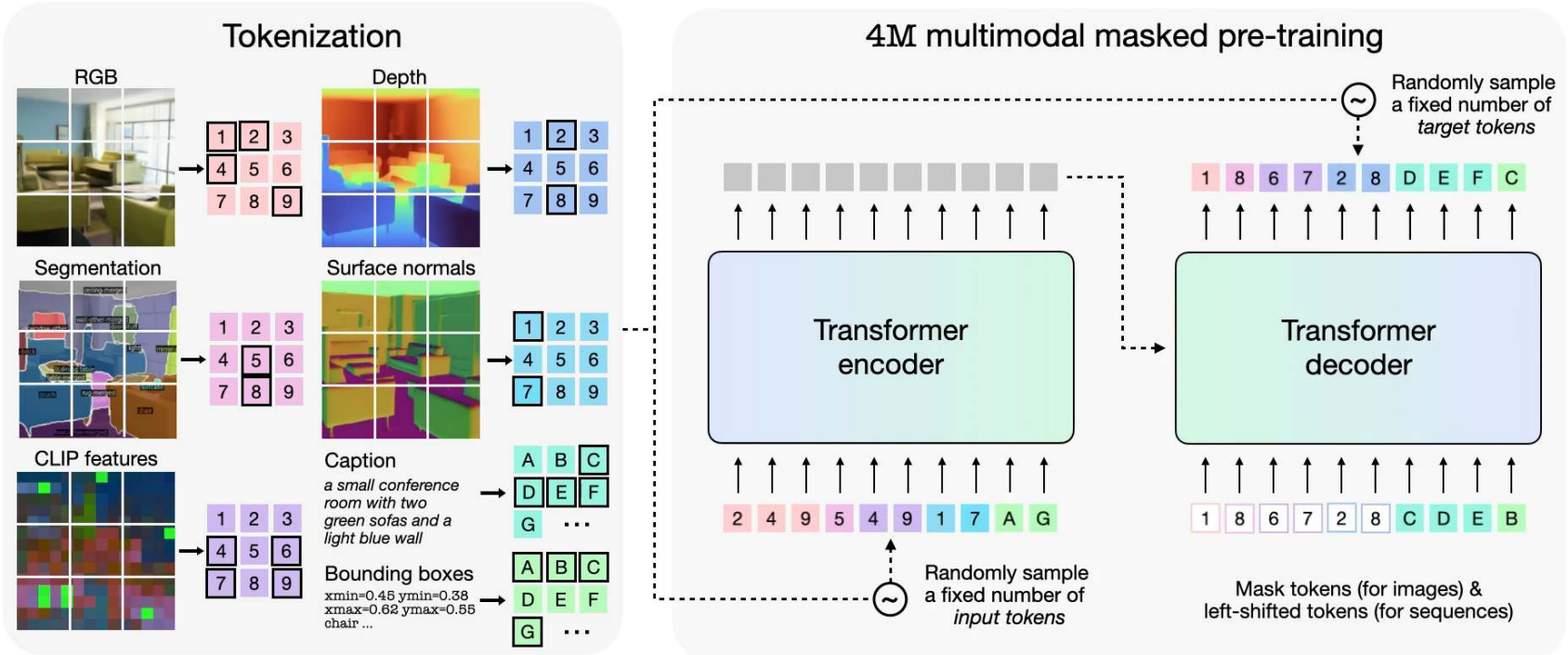
COM-304

Intelligent systems: communications & AI

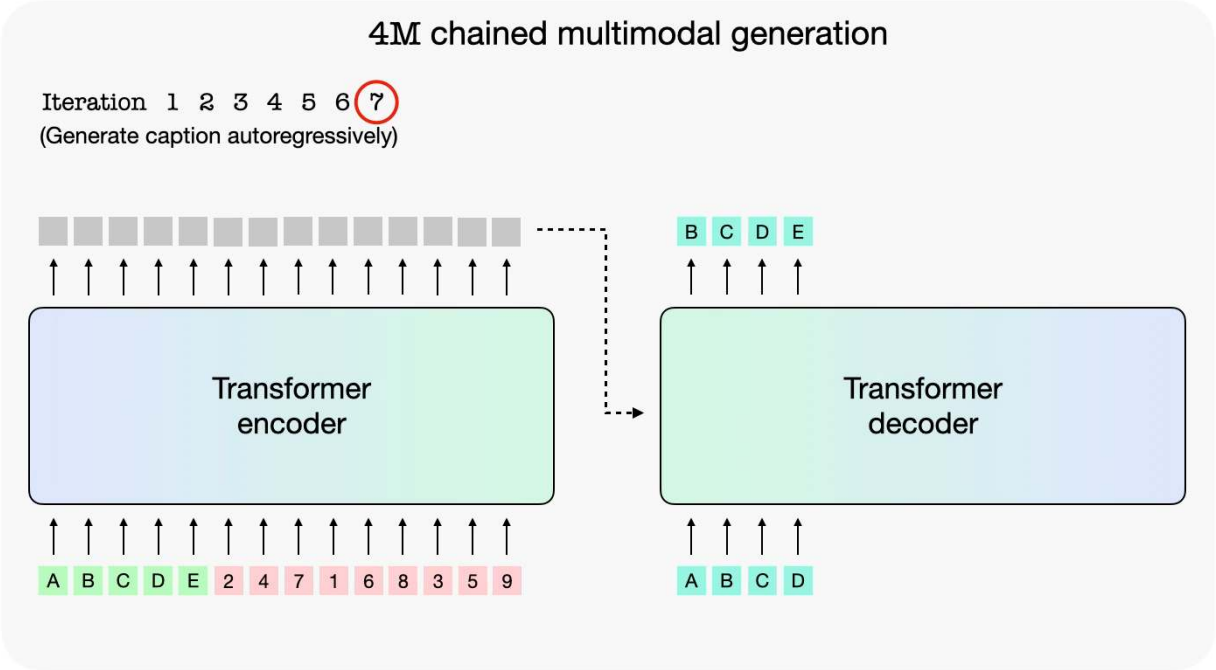
Jason Toskov

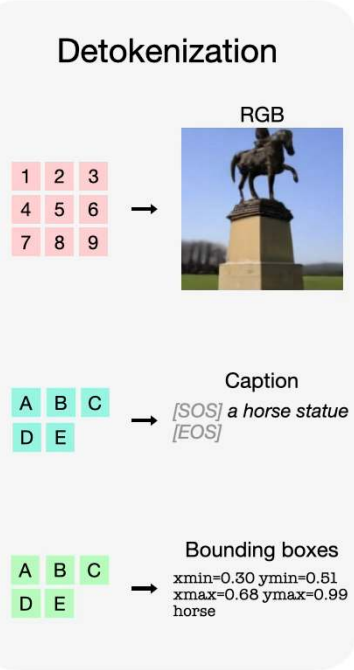
4M Tutorial Rundown

École polytechnique fédérale de Lausanne


Tokenization
RGB
Depth
Segmentation
Surface normals
CLIP features
Caption
a small conference room with two green sofas and a light blue wall
Bounding boxes
xmin=0.45 ymin=0.38 xmax=0.62 ymax=0.55 chair ...
4M multimodal masked pre-training
Randomly sample a fixed number of target tokens
Transformer encoder
Transformer decoder
Randomly sample a fixed number of input tokens
Mask tokens (for images) & left-shifted tokens (for sequences)

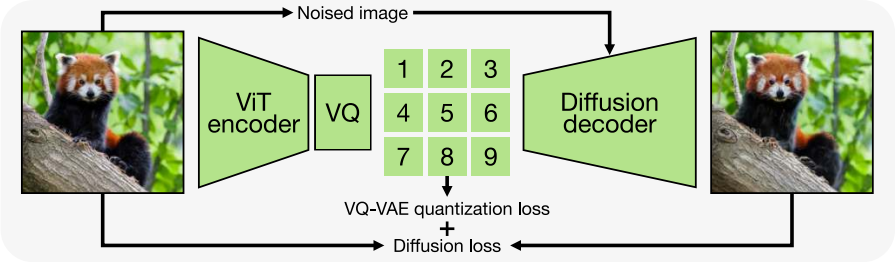
4M Generation



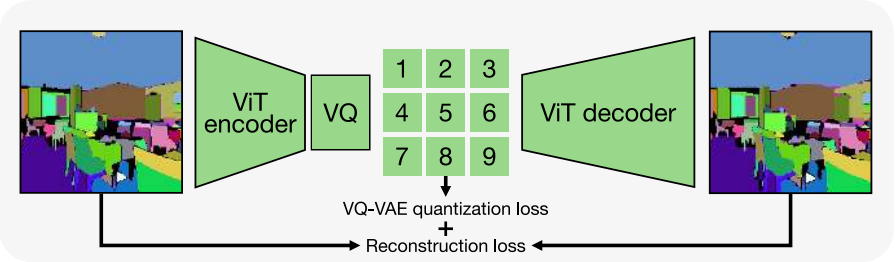


Tokenization

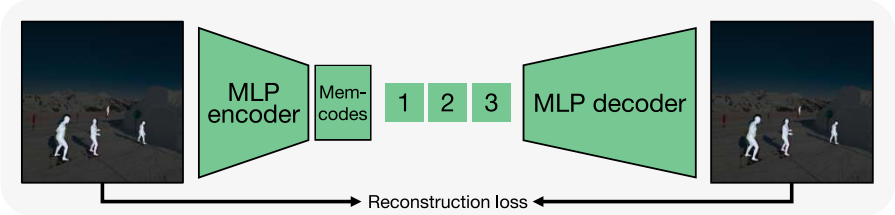
Spatial discrete VAE with diffusion decoder: RGB, normal, depth, edges



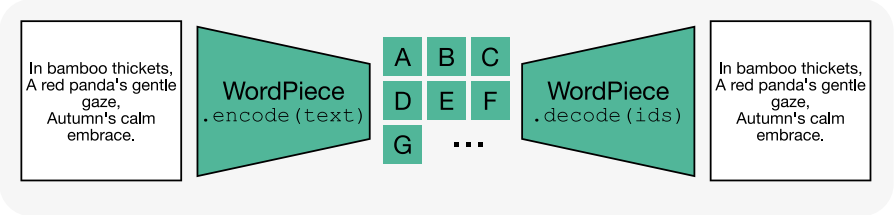
Spatial discrete VAE: Segmentation, CLIP, DINOv2, ImageBind, SAM inst.



MLP discrete VAE: Human poses, DINOv2 & ImageBind global tokens



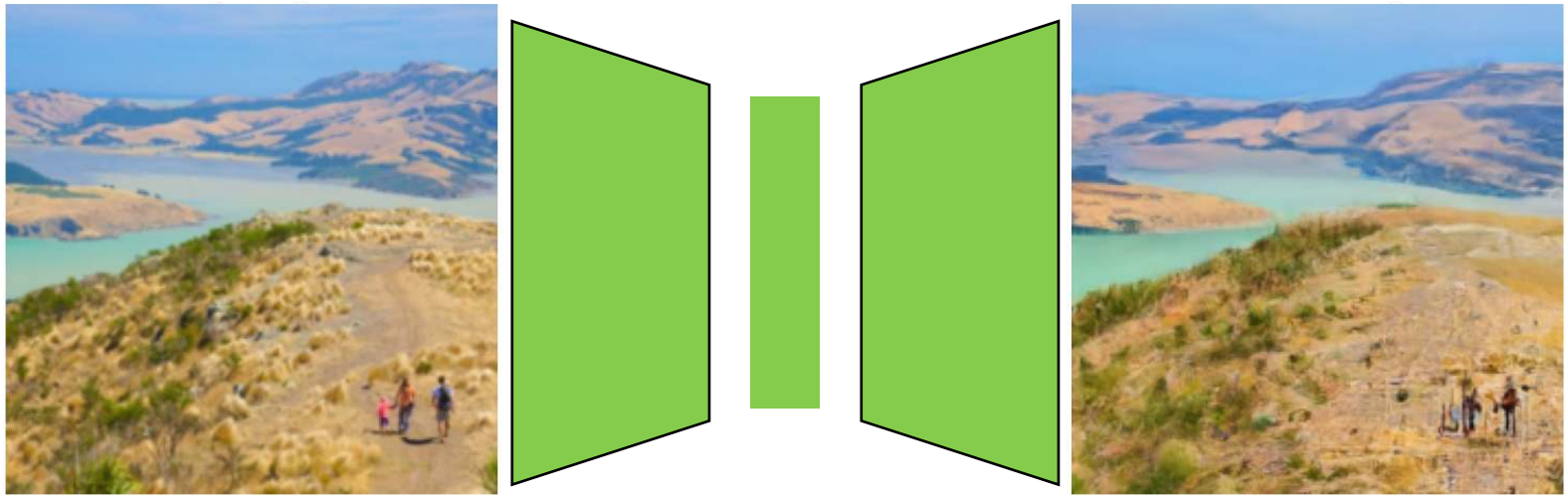
Sequence tokenizer: Text, bounding boxes, metadata, color palette



RGB Tokenization

Original Image

Reconstructed Image



Effects of Resolution

Effects of Resolution


```python
resolutions = [224, 256, 288, 320, 352, 384, 416, 448]
fig, axes = plt.subplots(2, 4, figsize=(16, 8))

# Loop over resolutions
for idx, res in enumerate(resolutions):
    # Resize image to correct resolution
    img_resized = center_crop(img_pil, (min(img_pil.size), min(img_pil.size))).resize((res, res))

    # Preprocess image for tokenization + add batch dimension
    img = rgb_transform.postprocess(img_resized).unsqueeze(0).to(device)

    # Tokenize & decode
    tokenized_rgb = toks['tok_rgb'].tokenize(img)
    reconstructed_rgb = toks['tok_rgb'].decode_tokens(tokenized_rgb, image_size=res, timesteps=19)

    # Denormalize to make visualizable
    reconstructed_rgb = denormalize(reconstructed_rgb, mean=IMAGENET_INCEPTION_MEAN, std=IMAGENET_INCEPTION_STD)[0]
    reconstructed_rgb = reconstructed_rgb.cpu().permute(1, 2, 0)

    # Plotting
    ax = axes[idx // 4, idx % 4]
    ax.imshow(reconstructed_rgb)
    ax.set_title(f"Résolution: {res}x{res}")
    ax.axis("off")

plt.show()
```

Effects of Resolution

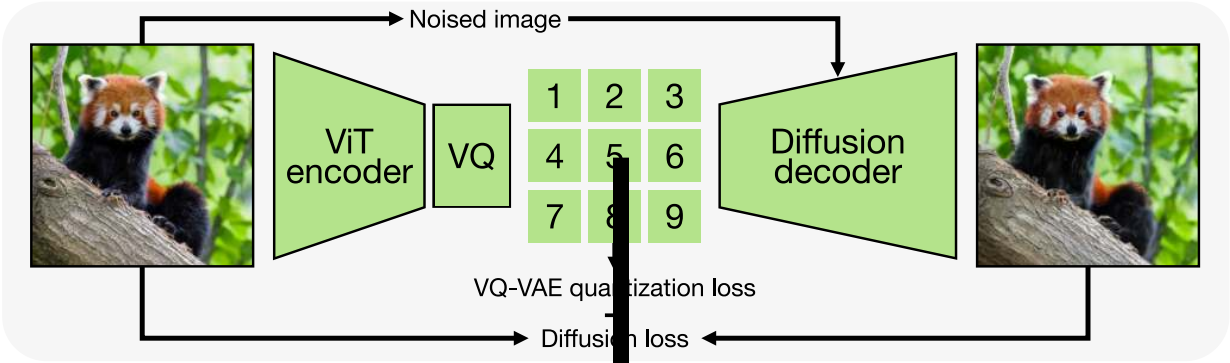
Resolution: 512x512

Resolution: 576x576

Resolution: 640x640

Token Values

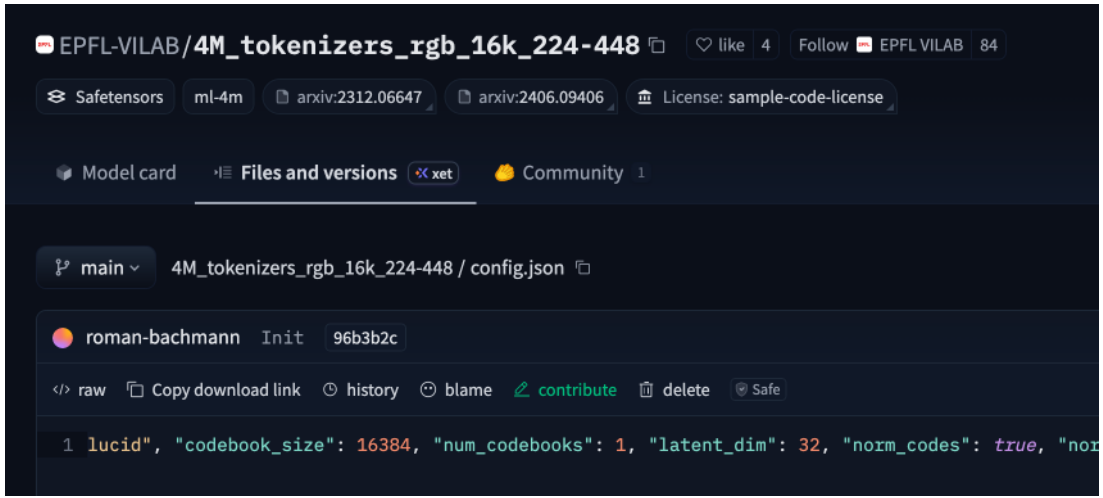
Spatial discrete VAE with diffusion decoder: RGB, normal, depth, edges



[?,?]

Token Values


```
toks = {
    'tok_rgb': DiVAE.from_pretrained('EPFL-VILAB/4M_tokenizers_rgb_16k_224-448').eval().to(device),
    'tok_depth': DiVAE.from_pretrained('EPFL-VILAB/4M_tokenizers_depth_8k_224-448').eval().to(device
```



[0,16383]

Effects of Diffusion Steps

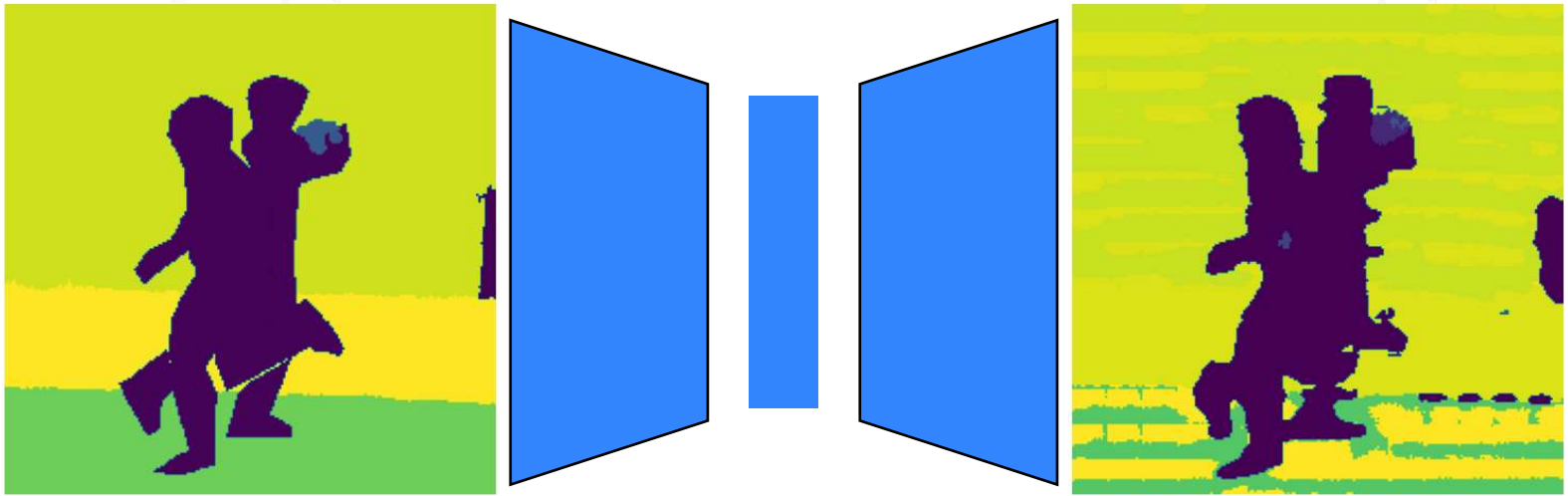
Effects of Diffusion Steps


```
timesteps_list = [1, 10, 20, 50, 100]
resolutions = [224, 448]

fig, axes = plt.subplots(len(resolutions), len(timesteps_list), figsize=(15, 6))
# Loop over resolutions
for row, res in enumerate(resolutions):
    img_resized = center_crop(img_pil, (min(img_pil.size), min(img_pil.size))).resize((res, res))
    img = rgb_transform.postprocess(img_resized).unsqueeze(0).to(device)

    tokenized_rgb = toks['tok_rgb'].tokenize(img)

    # And over timesteps
    for col, timesteps in enumerate(timesteps_list):
        reconstructed_rgb = toks['tok_rgb'].decode_tokens(
            tokenized_rgb, image_size=res, timesteps=timesteps
        )

        reconstructed_rgb = denormalize(
            reconstructed_rgb, mean=IMAGENET_INCEPTION_MEAN, std=IMAGENET_INCEPTION_STD
        )[0]
        reconstructed_rgb = reconstructed_rgb.cpu().permute(1, 2, 0)

        ax = axes[row, col]
        ax.imshow(reconstructed_rgb)
        ax.set_title(f"Res: {res}x{res}, Steps: {timesteps}")
        ax.axis("off")
```

Semantic Segmentation Tokenization

Original Sem. Seg.

Reconstructed Sem. Seg.



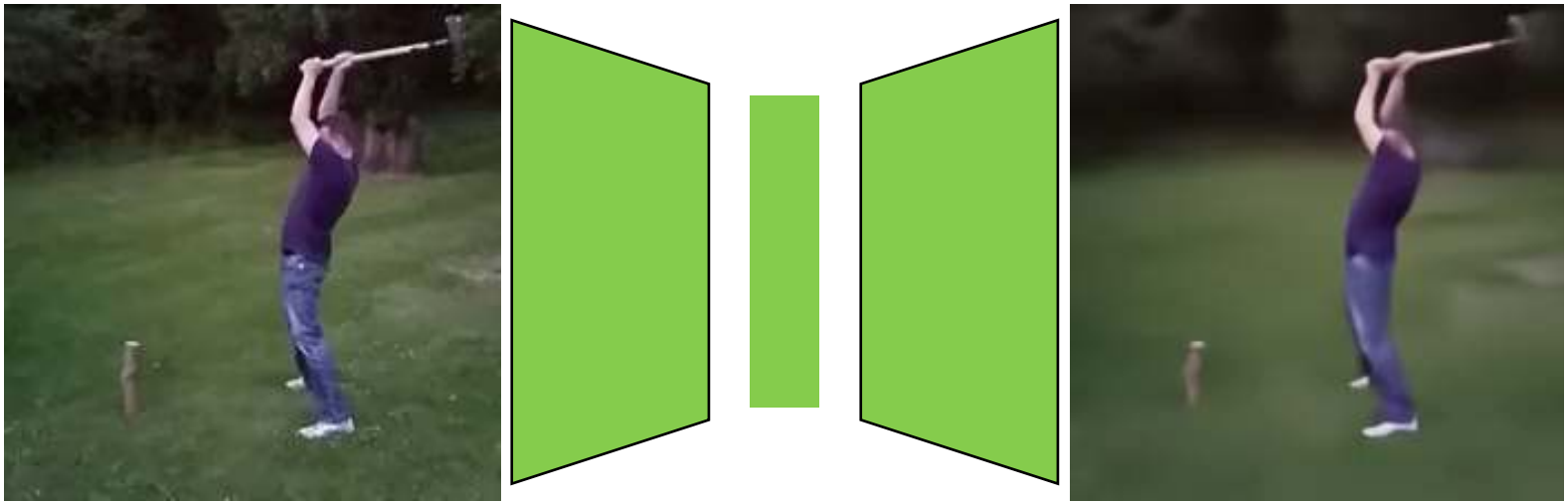
Semantic Segmentation Tokenization


```python
# Proper processing for semseg images
semseg_transform = SemsegTransform(shift_idx_by_one=True)

# Get the semseg
img_path = 'data/semseg_images_exercise_1/sample1.png'
img_pil = semseg_transform.load(img_path)
img_pil = semseg_transform.preprocess(img_pil)
img_pil = center_crop(img_pil, (min(img_pil.size), min(img_pil.size))).resize((224, 224))
img = semseg_transform.postprocess(img_pil).unsqueeze(0).to(device)

# Tokenize and reconstruct with semseg tokenizer
tok_semseg = toks['tok_semseg']
tokenized_semseg = tok_semseg.tokenize(img)
reconstructed_semseg = tok_semseg.decode_tokens(tokenized_semseg, image_size=224, timesteps=19)

# Take the most likely class to get segmentation mask
predicted_mask = reconstructed_semseg.argmax(dim=1)  # Shape: (1, 224, 224)

# Plot our greyscale semseg image
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

axes[0].imshow(img[0].cpu().numpy(), cmap='viridis')
axes[0].set_title("Original Semseg Image")
axes[0].axis("off")

axes[1].imshow(predicted_mask[0].cpu().numpy(), cmap='viridis')
axes[1].set_title("Reconstructed Semseg Image")
axes[1].axis("off")

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

Video Tokenization

Original Video

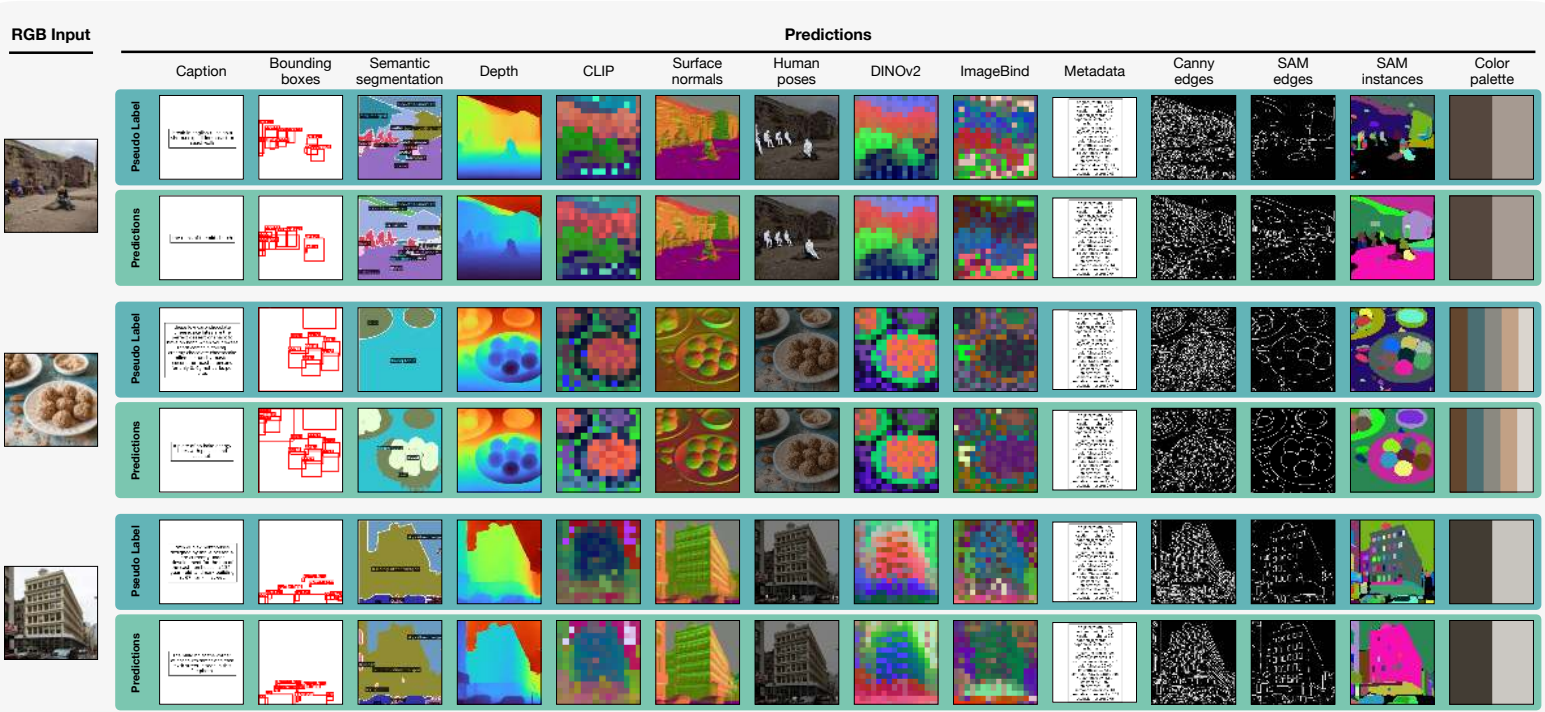
Reconstructed Video



Video Tokenization


```python
# Get frames
sampled_frames = load_sampled_frames('data/videos_exercise_1/sample1.mp4', step=10)

# Frames are just RGB images
rgb_transform = RGBTransform(imagenet_default_mean_and_std=False)

# Process each frame like before
processed_frames = []
for frame in sampled_frames:
    frame_pil = Image.fromarray(frame)
    frame_pil = rgb_transform.preprocess(frame_pil)
    frame_pil = center_crop(frame_pil, (min(frame_pil.size), min(frame_pil.size))).resize((224, 224))
    frame_tensor = rgb_transform.postprocess(frame_pil).unsqueeze(0).to(device)
    processed_frames.append(frame_tensor)
video_tensor = torch.cat(processed_frames, dim=0)

# Frames are now batched images
tokenized_video = toks['tok_rgb'].tokenize(video_tensor)
reconstructed_video = toks['tok_rgb'].decode_tokens(tokenized_video, image_size=224, timesteps=50)

# Save videos
saved_path_original = save_video_tensor(
    video_tensor, 'data/videos_exercise_1/sample1.mp4', save_name='original'
)
saved_path_reconstructed = save_video_tensor(
    reconstructed_video, 'data/videos_exercise_1/sample1.mp4', save_name='reconstructed'
)
```

Generation from RGB



Generation Scheduling


```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation Scheduling


```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation Scheduling


```
tokens_per_target_dict = {
    'tok_rgb@224': 196,
    'tok_clip@224': 196,
    'tok_dinov2@224': 256,
    'tok_imagebind@224': 256,
    'tok_depth@224': 196,
    'tok_normal@224': 196,
    'tok_semseg@224': 196,
    'tok_canny_edge@224': 196,
    'tok_sam_edge@224': 196,
    'caption': 256,
    'det': 256,
    'human_poses': 275,
    'sam_instance': 256,
    'color_palette': 23,
    'metadata': 40
}
```

Generation Scheduling


```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation Scheduling

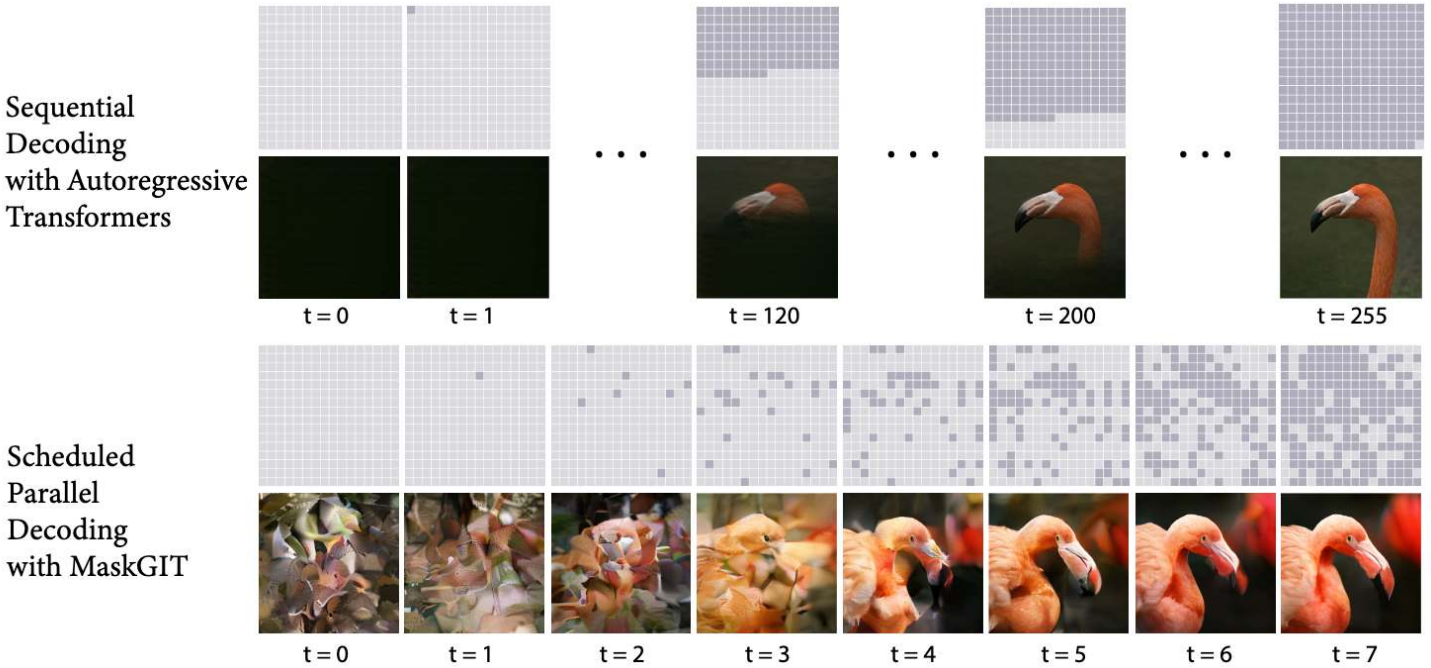

```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation Scheduling

```
autoregression_schemes_dict = {
    'tok_rgb@224': 'roar',
    'tok_clip@224': 'roar',
    'tok_dinov2@224': 'roar',
    'tok_imagebind@224': 'roar',
    'tok_depth@224': 'roar',
    'tok_normal@224': 'roar',
    'tok_semseg@224': 'roar',
    'tok_canny_edge@224': 'roar',
    'tok_sam_edge@224': 'roar',
    'caption': 'autoregressive',
    'det': 'autoregressive',
    'human_poses': 'autoregressive',
    'sam_instance': 'autoregressive',
    'color_palette': 'autoregressive',
    'metadata': 'autoregressive'
}
```

- Autoregressive: 1 token at a time
- ROAR: ***k*** tokens per step **randomly** sampled
- MaskGIT: ***k*** tokens per step sampled based on **probability**

Generation Scheduling



[MaskGIT: Masked Generative Image Transformer, Chang et al. 2022]

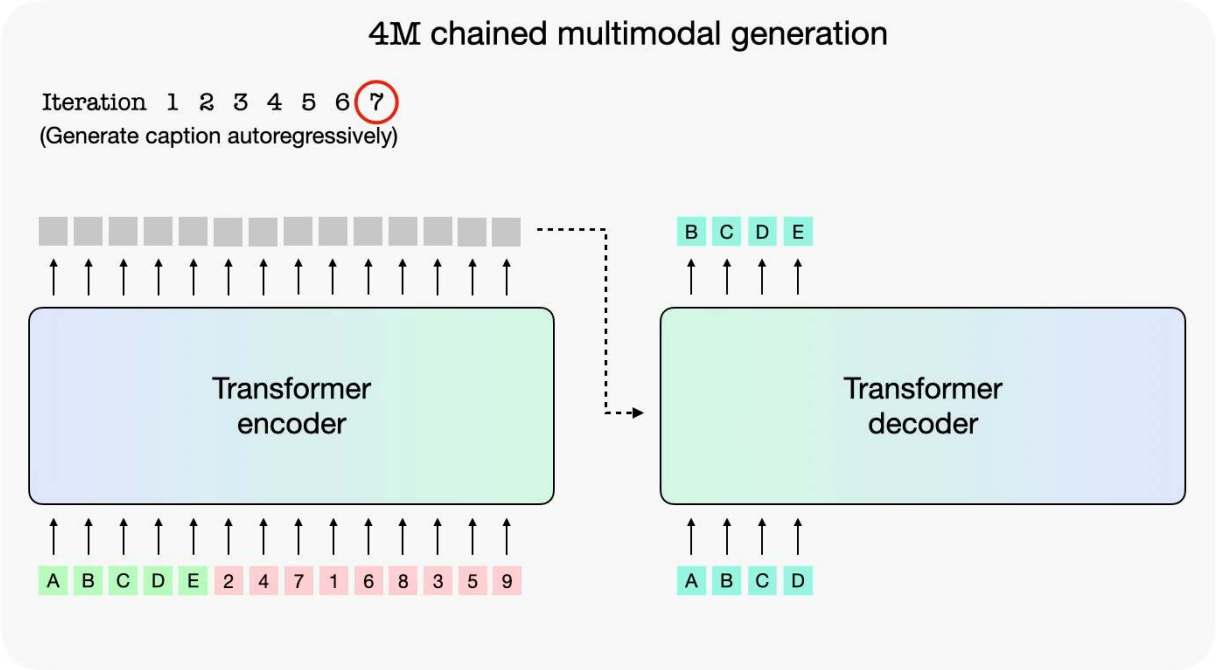
Generation Scheduling


```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

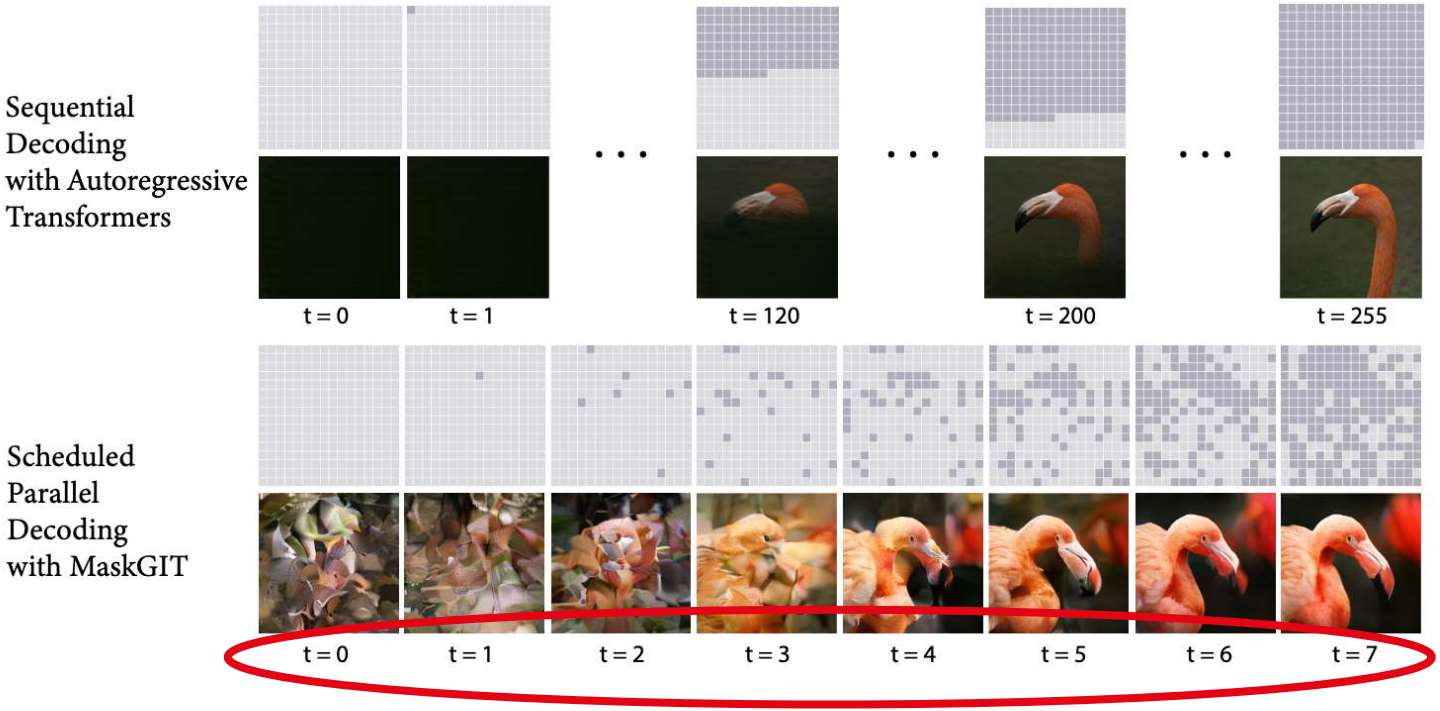
Generation Scheduling


```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation Scheduling



Generation Scheduling



[MaskGIT: Masked Generative Image Transformer, Chang et al. 2022]

Generation Scheduling

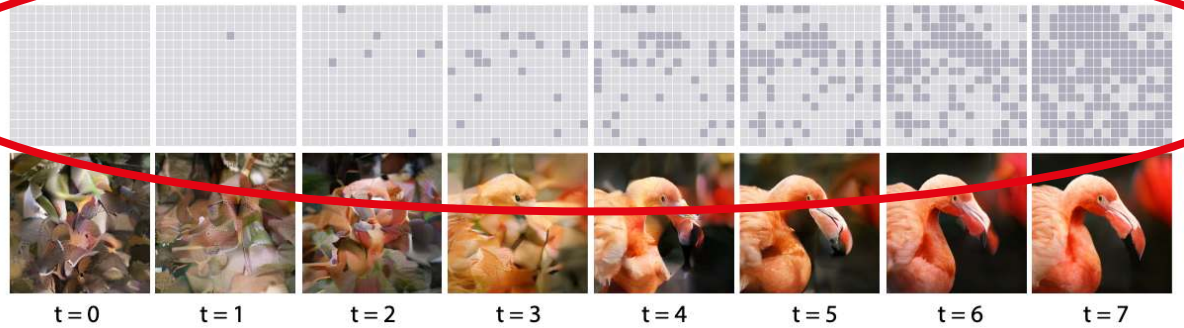

```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

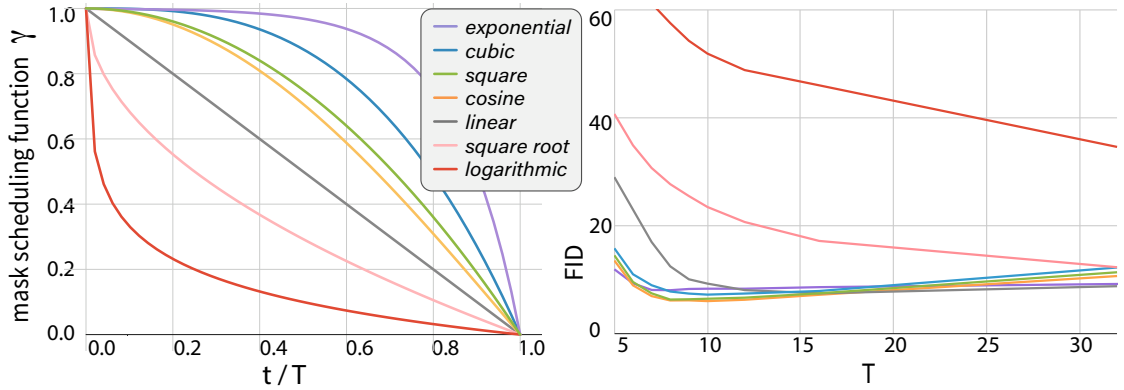
Generation Scheduling


```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation Scheduling

Scheduled Parallel Decoding with MaskGIT





[MaskGIT: Masked Generative Image Transformer, Chang et al. 2022]

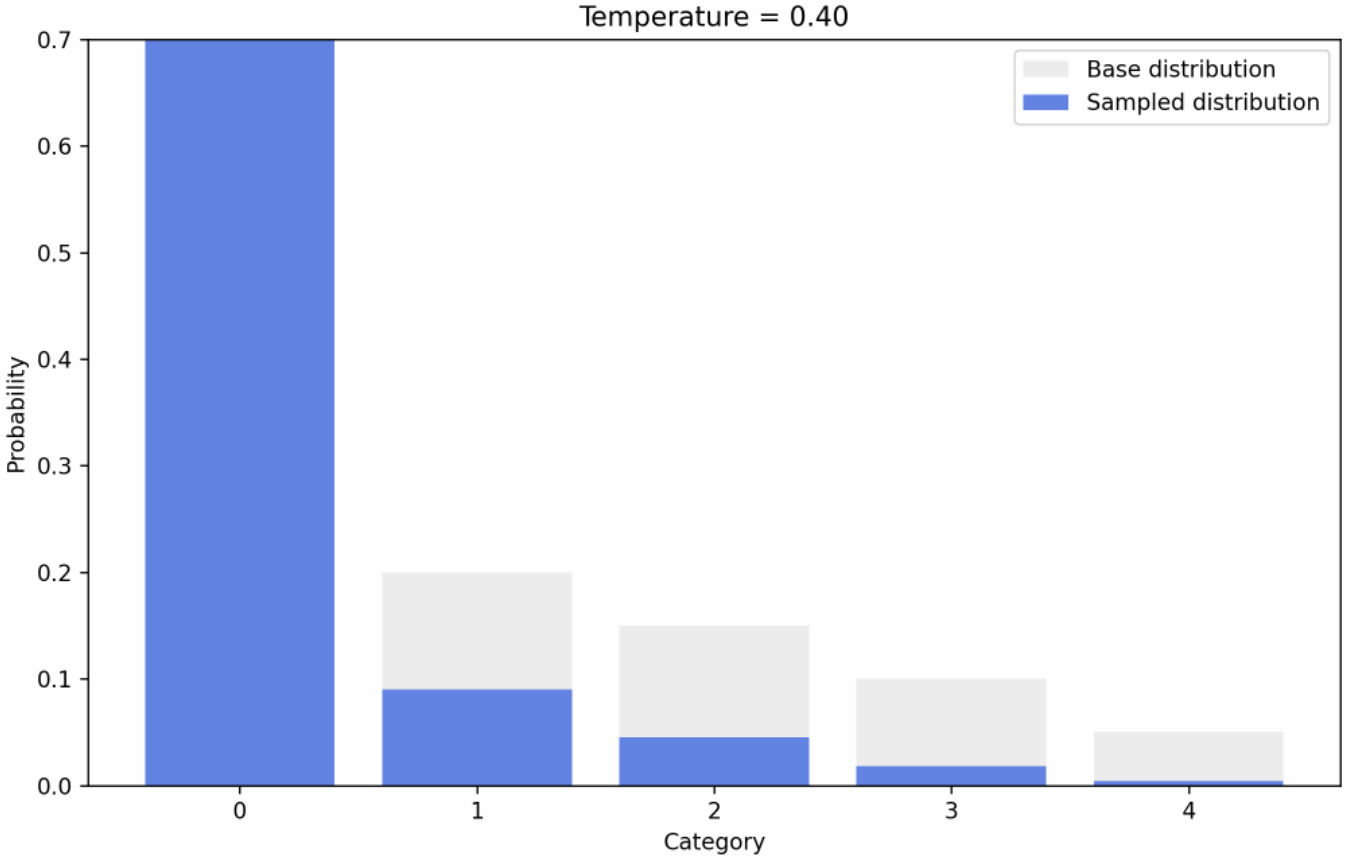
Generation Scheduling


```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation Scheduling


```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation Scheduling



Generation Scheduling


```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation Scheduling

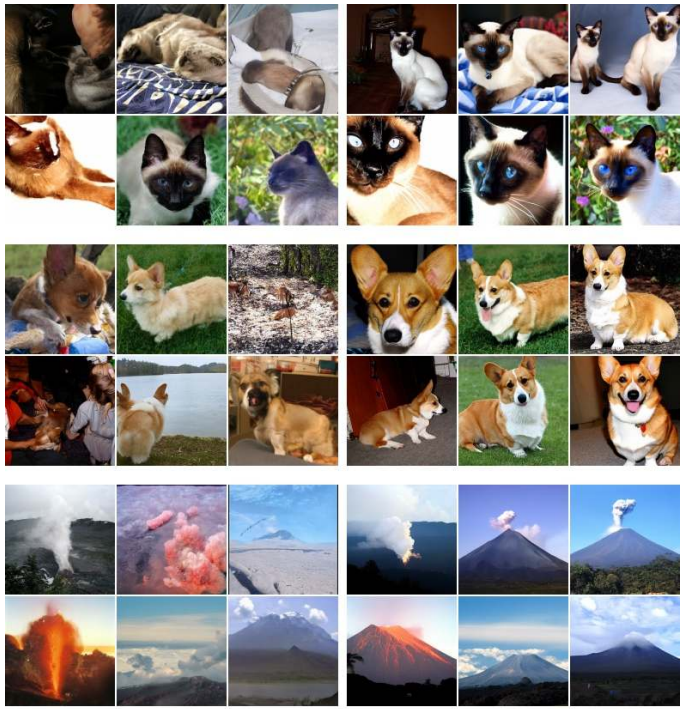

```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation Scheduling


```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation Scheduling

No Guidance Guidance



Classifier-free guidance. Classifier-free guidance is crucial for improving both image fidelity and how well the generation matches the conditioning. It is most commonly used in diffusion models [50], but can be applied in token-based models as well [40, 123, 18]. We perform classifier-free guidance by computing a weighted combinations of the logits of a forward pass with the conditioning and one without the conditioning:

$$\text{logits}_{\text{guided}} = \text{logits}_{\text{uncond}} + w\,(\text{logits}_{\text{cond}} - \text{logits}_{\text{uncond}})\,.$$

Here, w is the guidance scale. When performing chained generation, we add each fully generated modality to the set of guided modalities.

Multimodal guidance. While guidance has been shown to significantly improve image quality, it can still happen that generative models ignore parts of the input, unpredictably focus on some parts more than others, or generate undesired concepts. Negative prompting [78] is a popular way of keeping the model from generating undesired concepts. Liu et al. [68] show that performing compositional guidance on multiple conditions can further improve text-image similarity. In a similar way, we can perform compositional generation by weighting different (parts of) modalities by different continous amounts – even negatively. We can do this by computing a weighted sum of the logits of an unconditional case and the logits of each conditional case:

$$\text{logits}_{\text{guided}} = \text{logits}_{\text{uncond}} + \sum_{i=1}^{n} w_i \left(\text{logits}_{\text{cond, i}} - \text{logits}_{\text{uncond}}\right)\,.$$

Generation Scheduling


```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation Scheduling


```python
def create_generation_schedule_rgb_to_others(
    target_domains, decoding_steps, temps,
    cfg_scales, img, cfg_grow_conditioning=True
):
    """
    Create RGB to any other conditional domain
    """

    cond_domain = ['rgb@224']
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = cfg_schedules
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ['linear' if 'tok_' in target_domain else None for target_domain in target_domains]
    schedule = build_chained_generation_schedules(
        cond_domains=cond_domain,
        target_domains=target_domains,
        tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes,
        decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules,
        temps=temps,
        temp_schedules=temp_schedules,
        cfg_scales=cfg_scales,
        cfg_schedules=cfg_schedules,
        cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {
        'rgb@224': {
            'tensor': img, # Batched tensor!
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domain:
        batched_sample = init_full_input_modality(batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]"))

    return schedule, batched_sample
```

Generation from RGB

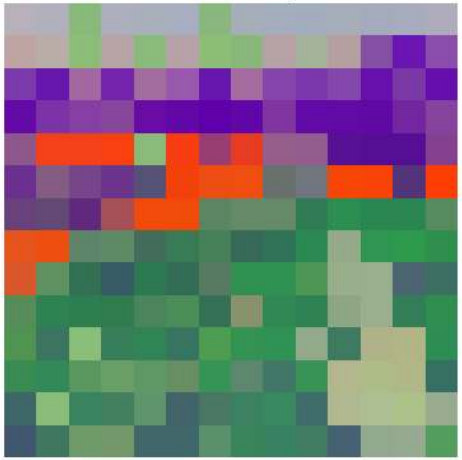

```python
# Example 1 usage: RGB to Caption

# Make the generation schedule (controls generation process)
schedule, batched_sample = create_generation_schedule_rgb_to_others(
    target_domains=['tok_clip@224'], # using CLIP as output
    decoding_steps=[1], # Put some steps
    temps=[0.1], # temperature value
    cfg_scales=[1.0], # Classifier-free guidance scale
    img=img,
)

# Run generation with standard top_p and top_k
out_dict = sampler.generate(
    batched_sample, schedule, text_tokenizer=text_tok, verbose=True, seed=0, top_p=0.8, top_k=0.0,
)

# Detokenize and convert to plottable format
dec_dict = decode_dict(
    out_dict, toks, text_tok, image_size=224, patch_size=16, decoding_steps=50
)

# Plotting
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8,4), facecolor=(1, 1, 1))

ax[0].imshow(img_pil)
ax[0].set_title('RGB input', fontsize=18)

ax[1].imshow(dec_dict['tok_clip@224'])
ax[1].set_title('CLIP Tokens pred.', fontsize=18)

for axis in ax.flatten():
    axis.set_axis_off()

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

Generation from RGB


```
# Then just change the target domains
target_domains=['tok_clip@224'], # CLIP
```

RGB input

CLIP Tokens pred.

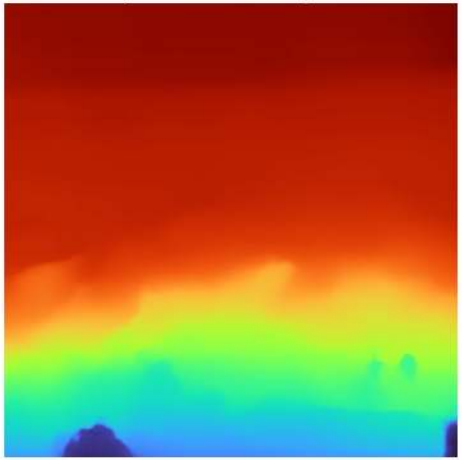


Generation from RGB


```
# Then just change the target domains
target_domains=['tok_depth@224'], # Depth
```

RGB input

Depth Tokens pred.



Generation from RGB


```
# Then just change the target domains
target_domains=['metadata'], # Metadata

# Plotting is different for metadata
metadata_pred = ',\n'.join([
    f'{k}: {v:.2f}' if isinstance(v, float) else f'{k}: {v}'
    for k, v in dec_dict['metadata'].items()
])
plot_text_in_square(ax[1], metadata_pred, wrap_width=36, fontsize=13)
```

RGB input

original_width: 1024,
original_height: 576,
caption_n_chars: 41,
caption_n_words: 8,
caption_n_sentences: 1, n_humans: 3,
n_sam_instances: 59,
n_coco_instances: 3,
coco_instance_diversity: 1,
colorfulness: 48.00, brightness:
147.90, contrast: 27.94, saturation:
102.00, entropy: 8.40, walkability:
0.00, objectness: 0.00,
semantic_diversity: 6,
geometric_complexity: 0.03,
occlusion_score: 0.01

Generation from RGB


```
# Then just change the target domains
target_domains=['det'], # Bounding boxes

# And for bounding boxes
ax[1].imshow(visualize_bboxes(np.array(img_pil), dec_dict['det'][0],))
ax[1].set_title('Bounding boxes pred.', fontsize=18)
```

RGB input

Bounding boxes pred.

Generation from RGB


```
# Then just change the target domains
target_domains=['color_palette'], # Colour palettes
```

RGB input

Color palette pred.

Generation from RGB


```
# Then just change the target domains
target_domains=['tok_sam_edge@224'], # Edges derived from SAM model instances
```

RGB input

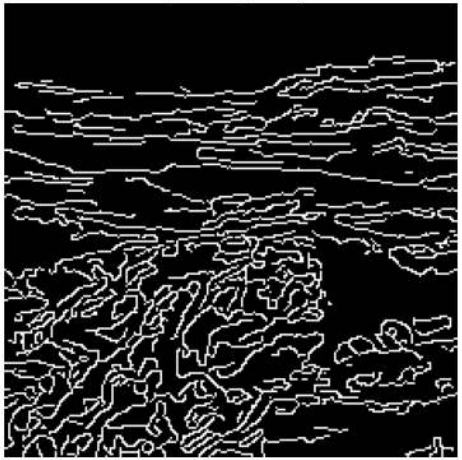
SAM edges pred.

Generation from RGB


```
# Then just change the target domains
target_domains=['tok_canny_edge@224'], # Canny edges
```

RGB input

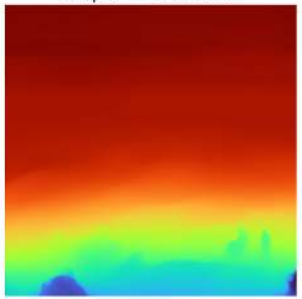
Canny edges pred.



Generation from RGB

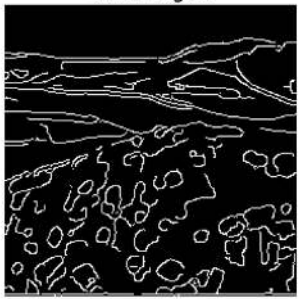
RGB Input

Depth Prediction



Bounding Boxes

SAM Edges

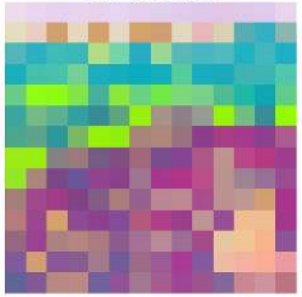


Canny Edges

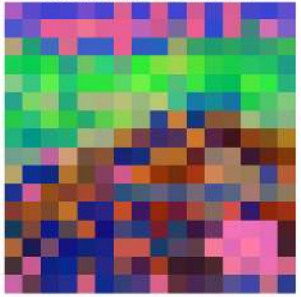
Metadata

original_width: 1024,
original_height: 576,
caption_n_chars: 60,
caption_n_words: 11,
caption_n_sentences: 1, n_humans: 3,
n_sam_instances: 61,
n_coco_instances: 3,
coco_instance_diversity: 1,
colorfulness: 48.00, brightness:
147.90, contrast: 27.94, saturation:
102.00, entropy: 8.60, walkability:
0.60, objectness: 0.00,
semantic_diversity: 8,
geometric_complexity: 0.03,
occlusion_score: 0.01

CLIP Features



ImageBind Features



DINOv2 Features

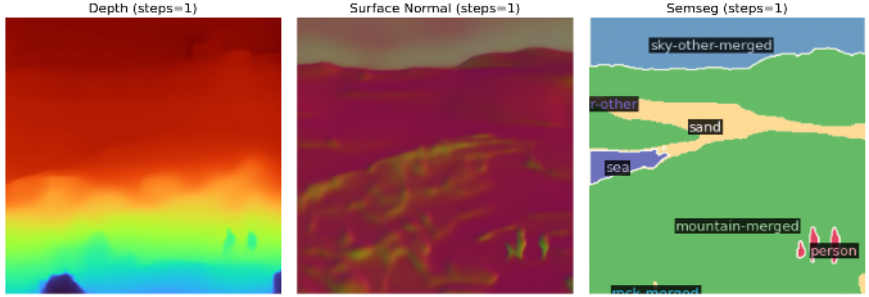
Color Palette

Parameter Ablation

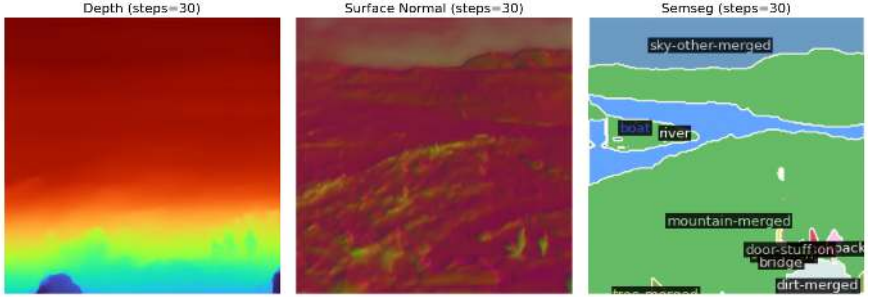

```python
# Keep other values fixed
target_domains = ["tok_depth@224", "tok_normal@224", "tok_semseg@224"]
temps = 0.01
cfg_scale = 2.0
top_p = 0.8
top_k = 0.0

# Loop over the ablated value
decoding_steps_list = [1, 10, 30, 50]
for decoding_steps in decoding_steps_list:
    dec_dict_all = {}

    # Generate for each target domain separately (no chaining)
    for target in target_domains:
        schedule, batched_sample = create_generation_schedule_rgb_to_others(
            target_domains=[target],
            decoding_steps=[decoding_steps],
            temps=[temps],
            cfg_scales=[cfg_scale],
            img=img,
            cfg_grow_conditioning=False
        )

        out_dict = sampler.generate(
            batched_sample, schedule, text_tokenizer=text_tok, seed=0, top_p=top_p, top_k=top_k
        )

        dec_dict = decode_dict(
            out_dict, toks, text_tok,image_size=224, patch_size=16, decoding_steps=decoding_steps
        )
        dec_dict_all[target] = dec_dict[target]
```

Parameter Ablation (steps)

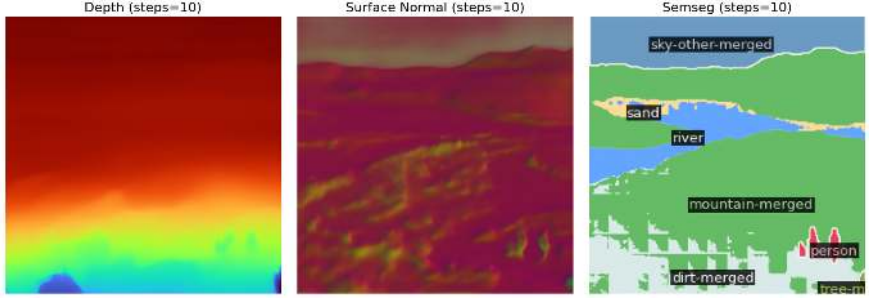
Steps = 1



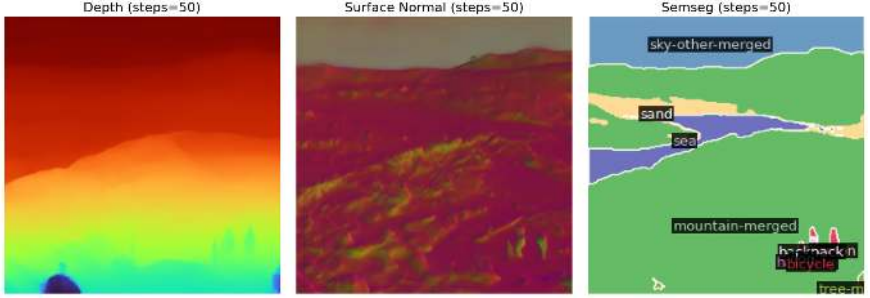
Steps = 30



Steps = 10

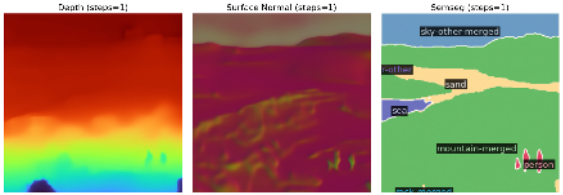


Steps = 50

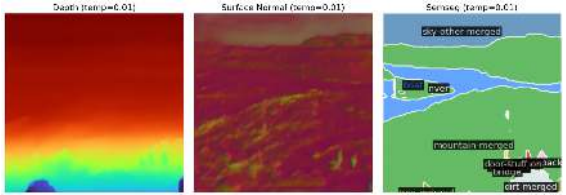


Parameter Ablation (Temperature)

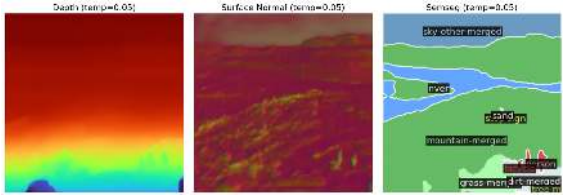
Temp = 0.0



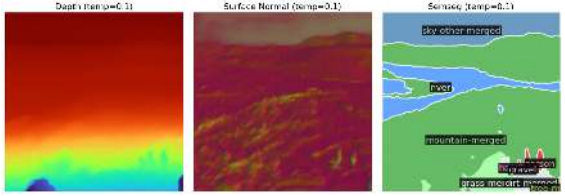
Temp = 0.01



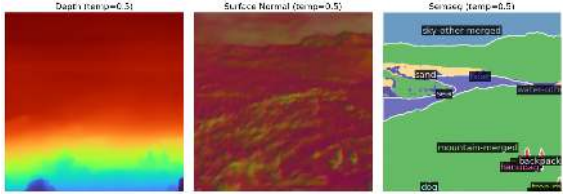
Temp = 0.05



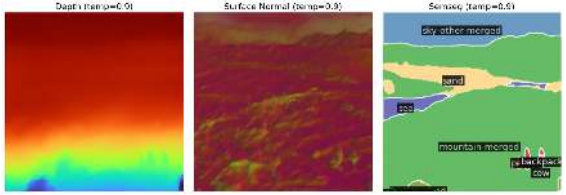
Temp = 0.1



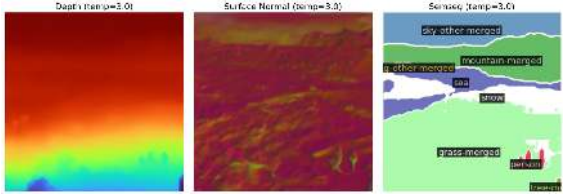
Temp = 0.5



Temp = 0.9



Temp = 3.0



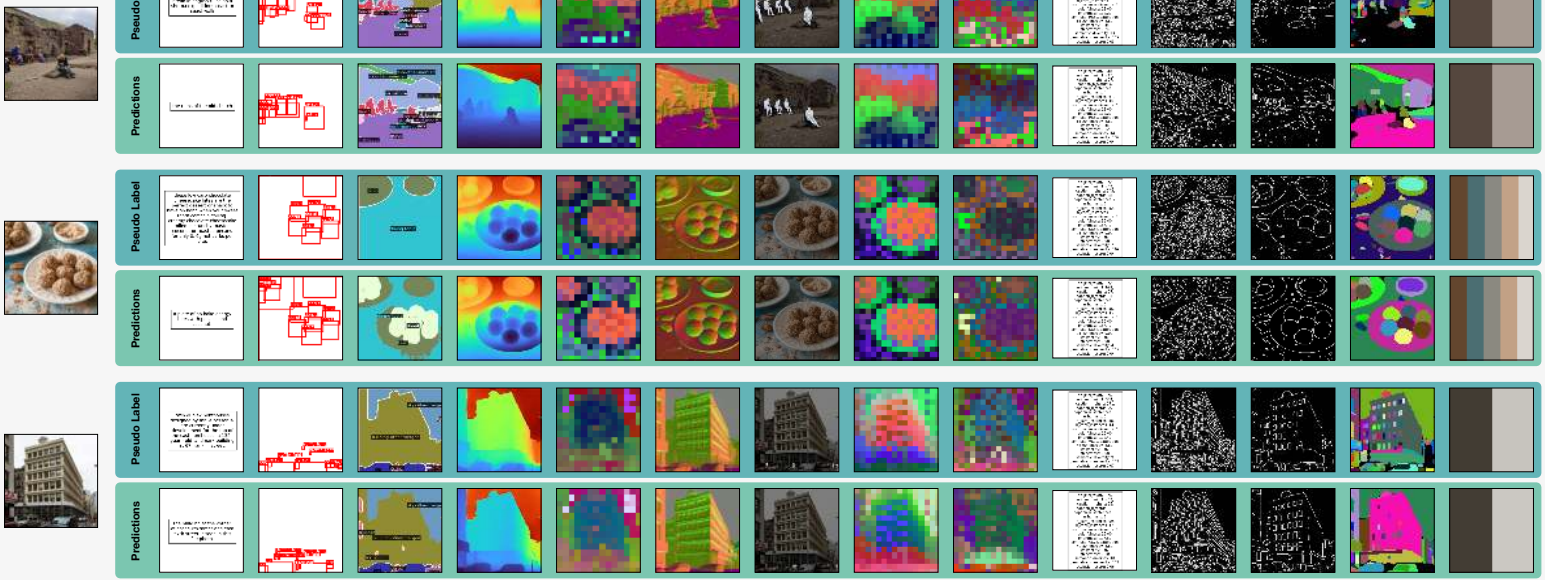
Parameter Ablation (Top P)

```python
captions = {}

temps = 0.5
decoding_steps = None
cfg_scale = 2.0

# Loop over top p values
top_p_list = [0.0, 0.01, 0.2, 0.5, 0.7, 0.9, 0.95, 1]
for top_p in top_p_list:

    schedule, batched_sample = create_generation_schedule_rgb_to_others(
        target_domains=["caption"],
        decoding_steps=[decoding_steps],
        temps=[temps],
        cfg_scales=[cfg_scale],
        img=img,
        cfg_grow_conditioning=False
    )

    out_dict = sampler.generate(
        batched_sample, schedule, text_tokenizer=text_tok,
        verbose=True, seed=None,
        top_p=top_p, top_k=0.0
    )

    dec_dict = decode_dict(out_dict, toks, text_tok, decoding_steps=decoding_steps)
    caption = dec_dict["caption"][0]

    captions[top_p] = caption

    print(f"Caption generated with top_p={top_p}: {caption}\n")
```

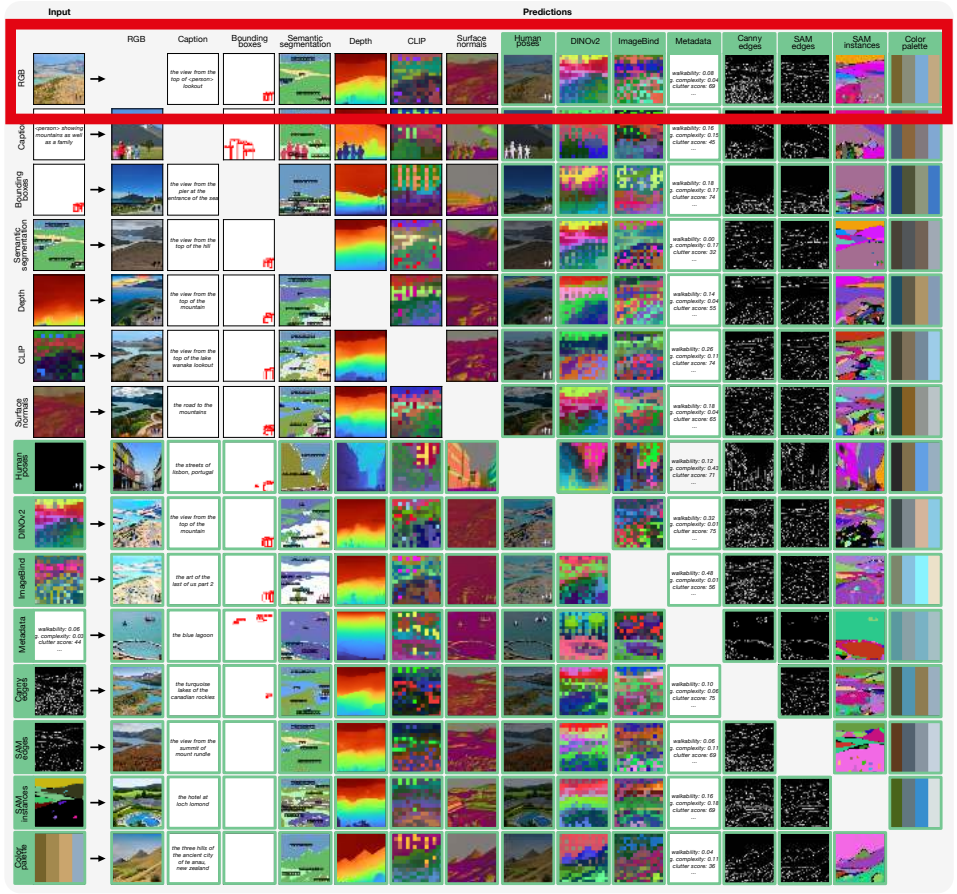
Parameter Ablation (Top P)

```
1it [00:00,  5.21it/s]
Caption generated with top_p=0.0: hiking the san bernardino national park

1it [00:00,  7.48it/s]
Caption generated with top_p=0.01: hikers on the road

1it [00:00,  7.47it/s]
Caption generated with top_p=0.2: hikers on the road

1it [00:00,  7.24it/s]
Caption generated with top_p=0.5: hikers on the road

1it [00:00,  5.78it/s]
Caption generated with top_p=0.7: hikers on the road to lake

1it [00:00,  4.06it/s]
Caption generated with top_p=0.9: hikers on the road to < person >

1it [00:00,  5.82it/s]
Caption generated with top_p=0.95: hiking the < person >

1it [00:00,  5.78it/s]
Caption generated with top_p=1.0: hikers on the dirt road
```

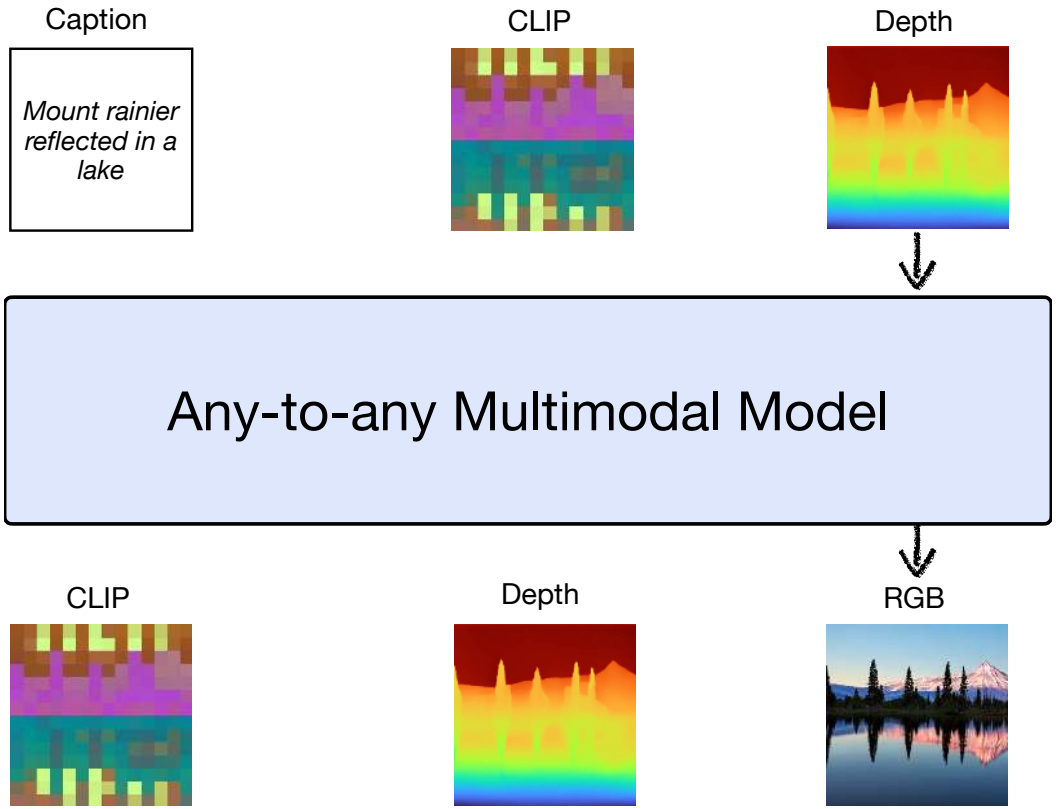
Generation from RGB



Chaining


Input
Predictions
RGB
Caption
Bounding boxes
Semantic segmentation
Depth
CLIP
Surface normals
Human poses
DINOv2
ImageBind
Metadata
Canny edges
SAM edges
SAM instances
Color palette

Chaining


Caption
Mount rainier reflected in a lake
CLIP
Depth
Any-to-any Multimodal Model
CLIP
Depth
RGB

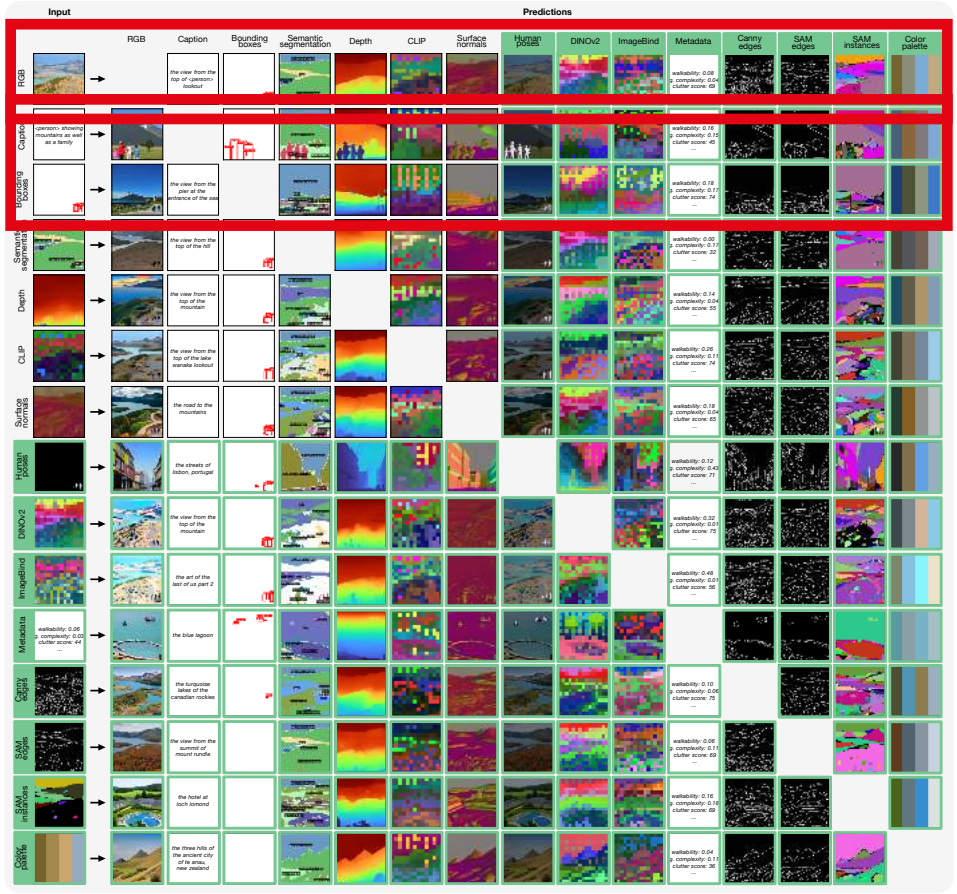
Sem. Seg. Evaluation with Chaining


```python
# More efficient: loop through configurations
chaining_configurations = [
    ["tok_depth@224", "tok_semseg@224"],
    ["tok_depth@224", "metadata", "tok_semseg@224"],
    ["tok_clip@224", "tok_dinov2@224", "tok_semseg@224"],
    ["metadata", "tok_depth@224", "tok_semseg@224"],
    ["tok_clip@224", "tok_semseg@224"],
    ["tok_dinov2@224", "tok_semseg@224"],
]

results = []
for elem in chaining_configurations:
    # Correct for different configuration lengths
    decoding_steps = [1] * len(elem)
    temps = [0.01] * len(elem)
    cfg_scales = [2.0] * len(elem)

    errors = []
    for batch, _ in tqdm(loader): # Loop over multiple samples to estimate mean
        batched_sample = { # Samples should be on the device
            'rgb@224': {k: v.to(device) for k, v in batch['rgb@224'].items()}
        }
        semseg_gt = batch['semseg_coco']['tensor'].unsqueeze(1).to(device)

        schedule, batched_sample = create_generation_schedule_rgb_to_others(
            target_domains=elem,
            decoding_steps=decoding_steps,
            temps=temps,
            cfg_scales=cfg_scales,
            img=batched_sample['rgb@224']['tensor'],
            cfg_grow_conditioning=True, # We are chaining -> grow conditioning
        )
        out_dict = sampler.generate(
            batched_sample, schedule, text_tokenizer=text_tok,
            verbose=False, seed=0,
            top_p=0.8, top_k=0.0,
        )
        dec_dict = decode_dict(
            out_dict, toks, text_tok,
            image_size=224, patch_size=16,
            decoding_steps=50, to_rgb=False
        )
        output_argmax = dec_dict['tok_semseg@224'].argmax(1).unsqueeze(1)

        for i in range(len(semseg_gt)):
            metrics_argmax = get_semseg_metrics(output_argmax[[i]], semseg_gt[[i]])
            errors.append({'argmax': metrics_argmax})

    # Get average performance for this configuration
    final_performance = [single_img_stats['argmax']['mean_iou'] for single_img_stats in errors]
    mean_mIoU = sum(final_performance) / len(final_performance) if final_performance else None

    results.append((elem, mean_mIoU))

print("\n Final results :")
for config, score in results:
    print(f"Configuration: {' -> '.join(config)} | mIoU: {score:.4f}")
```

Sem. Seg. Evaluation with Chaining


```
Final results :


Configuration: tok_depth@224 -> tok_semseg@224 | mIoU: 36.7818

Configuration: tok_depth@224 -> metadata -> tok_semseg@224 | mIoU: 35.3912

Configuration: tok_clip@224 -> tok_dinov2@224 -> tok_semseg@224 | mIoU: 38.7242

Configuration: metadata -> tok_depth@224 -> tok_semseg@224 | mIoU: 36.5050

Configuration: tok_clip@224 -> tok_semseg@224 | mIoU: 39.1254

Configuration: tok_dinov2@224 -> tok_semseg@224 | mIoU: 38.8018
```

Generation From Anything



Any to RGB Generation

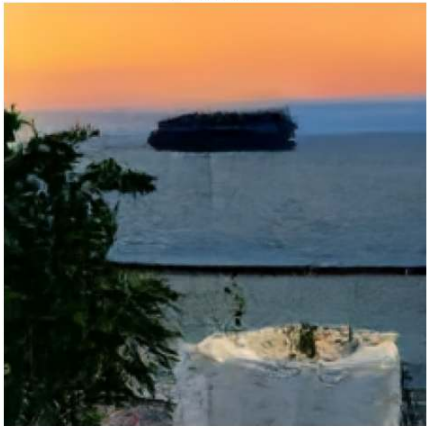

```python
# Loop over scenarios
scenarios = {
    "No Middle Modality": ["tok_rgb@224"],
    "With CLIP": ["tok_clip@224", "tok_rgb@224"],
    "With Color Palette": ["color_palette", "tok_rgb@224"],
    "With Meta Data": ["metadata", "tok_rgb@224"]
}

results = {}
for scenario_name, target_domains in scenarios.items():
    # Could also be looped over using different values per scenario
    decoding_steps = [50] * len(target_domains)
    temps = [3.0] * len(target_domains)
    cfg_scales = [2.0] * len(target_domains)

    # Get all default mappings
    cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
    temp_schedules = ["onex:0.5:0.5" for _ in target_domains]
    autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
    tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
    token_decoding_schedules = ["linear" if "tok_" in target_domain else None for target_domain in target_domains]

    schedule = build_chained_generation_schedules(
        cond_domains=["caption", "det"], # Our inputs are always caption and bounding boxes
        target_domains=target_domains, tokens_per_target=tokens_per_targets,
        autoregression_schemes=autoregression_schemes, decoding_steps=decoding_steps,
        token_decoding_schedules=token_decoding_schedules, temps=temps, temp_schedules=temp_schedules,
        cfg_scales=cfg_scales, cfg_schedules=cfg_schedules, cfg_grow_conditioning=cfg_grow_conditioning
    )

    batched_sample = {}
    for target_mod, ntoks in zip(target_domains, tokens_per_targets):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    batched_sample = custom_text(
        batched_sample, input_text=caption, eos_token="[EOS]",
        key="caption", device=device, text_tokenizer=text_tok
    )
    batched_sample = custom_text(
        batched_sample, input_text=bboxes, eos_token="[EOS]",
        key="det", device=device, text_tokenizer=text_tok
    )

    out_dict = sampler.generate(
        batched_sample, schedule, text_tokenizer=text_tok, verbose=True, seed=0, top_p=top_p, top_k=top_k
    )
    dec_dict = decode_dict( out_dict, toks, text_tok, image_size=224, patch_size=16, decoding_steps=50)

    results[scenario_name] = dec_dict["tok_rgb@224"] # We want to see the RGB generation
```

Any to RGB Generation

Toskov

No Middle Modality

With CLIP



With Color Palette

With Meta Data

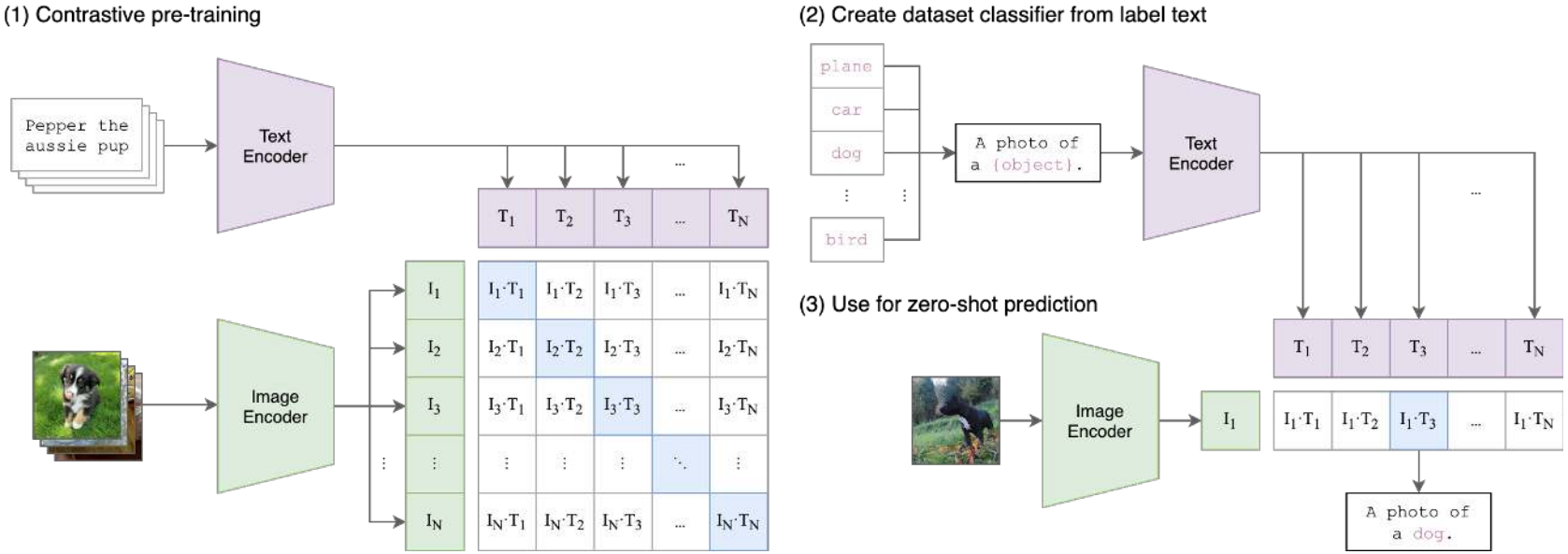
Any to RGB Generation (with Metadata)


```
metadata_transform = MetadataTransform(shuffle=False, random_trunc=False, return_chunks=False)
metadata_dict = {'semantic_diversity': 20}
metadata_str = metadata_transform.metadata_to_string(metadata_dict) + ' [S_1]'

target_domains = ["tok_rgb@224"] # Only RGB is the target (no chaining)
decoding_steps = [50]
temps = [3.0]
cfg_scales = [2.0]

cfg_schedules = [cfg_schedules_dict[target_domain] for target_domain in target_domains]
temp_schedules = ["onex:0.5:0.5" for _ in target_domains]
autoregression_schemes = [autoregression_schemes_dict[target_domain] for target_domain in target_domains]
tokens_per_targets = [tokens_per_target_dict[target_domain] for target_domain in target_domains]
token_decoding_schedules = ["linear" if "tok_" in target_domain else None for target_domain in target_domains]

schedule = build_chained_generation_schedules(
    cond_domains=["caption", "det", "metadata"], # Now we use metadata as an additional conditioning input
    target_domains=target_domains, tokens_per_target=tokens_per_targets, autoregression_schemes=autoregression_schemes,
    decoding_steps=decoding_steps, token_decoding_schedules=token_decoding_schedules, temps=temps, temp_schedules=temp_schedules,
    cfg_scales=cfg_scales, cfg_schedules=cfg_schedules, cfg_grow_conditioning=cfg_grow_conditioning
)

batched_sample = {}
for target_mod, ntoks in zip(target_domains, tokens_per_targets):
    batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

batched_sample = custom_text(
    batched_sample, input_text=caption, eos_token="[EOS]",
    key="caption", device=device, text_tokenizer=text_tok
)
batched_sample = custom_text(
    batched_sample, input_text=bboxes, eos_token="[EOS]",
    key="det", device=device, text_tokenizer=text_tok
)
batched_sample = custom_text(
    batched_sample, input_text=metadata_str, eos_token="[EOS]",
    key="metadata", device=device, text_tokenizer=text_tok
)

out_dict = sampler.generate(batched_sample, schedule, text_tokenizer=text_tok, seed=0, top_p=top_p, top_k=top_k)
dec_dict = decode_dict(out_dict, toks, text_tok, image_size=224, patch_size=16, decoding_steps=50)
```

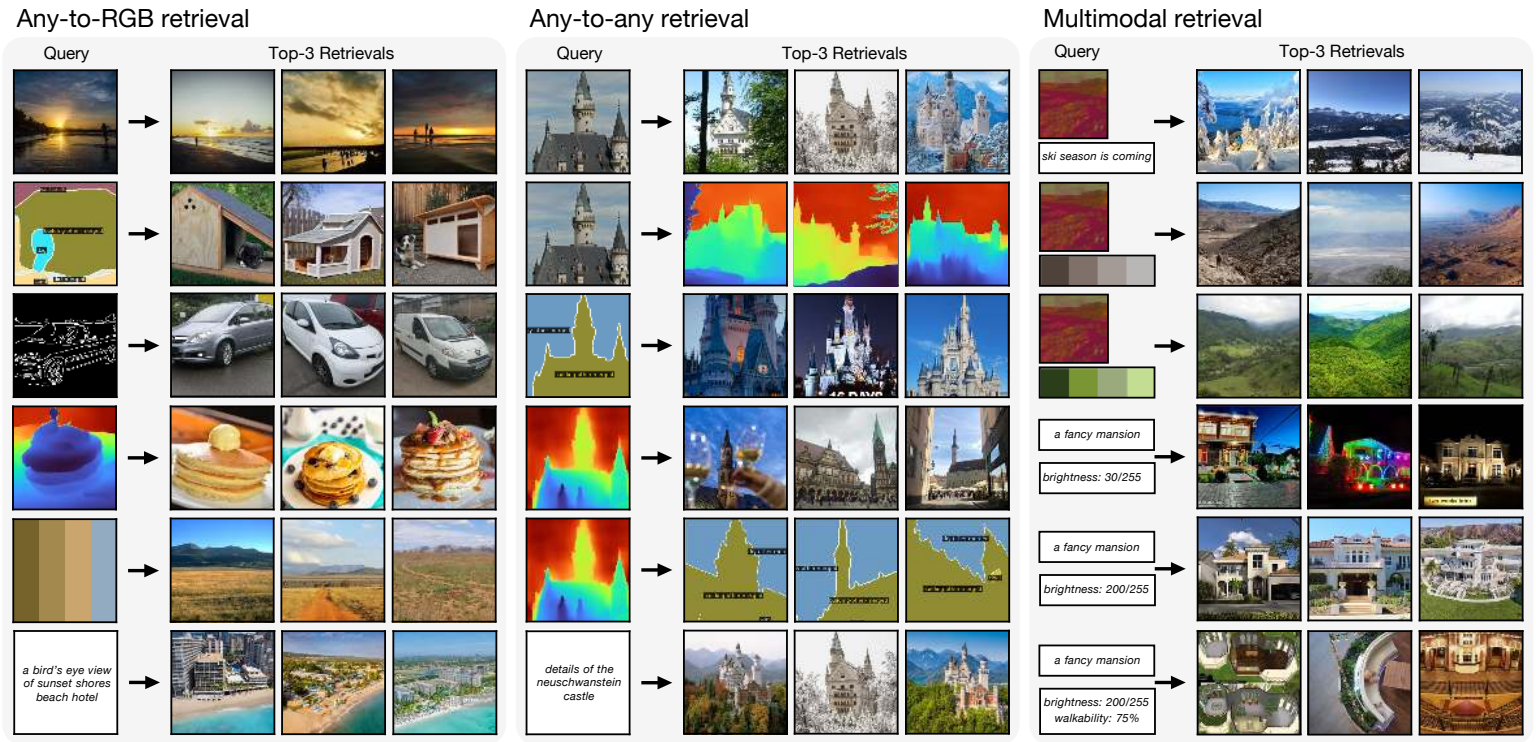
Any to RGB Generation (with Metadata)

with Metadata as third condition (No modality)

Retrieval


(1) Contrastive pre-training
Pepper the aussie pup
Text Encoder
Image Encoder
T_1 T_2 T_3 ... T_N
I_1 I_2 I_3 ... I_N
$I_1 \cdot T_1$ $I_1 \cdot T_2$ $I_1 \cdot T_3$ ... $I_1 \cdot T_N$
$I_2 \cdot T_1$ $I_2 \cdot T_2$ $I_2 \cdot T_3$ ... $I_2 \cdot T_N$
$I_3 \cdot T_1$ $I_3 \cdot T_2$ $I_3 \cdot T_3$ ... $I_3 \cdot T_N$
$I_N \cdot T_1$ $I_N \cdot T_2$ $I_N \cdot T_3$ ... $I_N \cdot T_N$
(2) Create dataset classifier from label text
plane
car
dog
bird
A photo of a {object}.
Text Encoder
(3) Use for zero-shot prediction
Image Encoder
I_1
T_1 T_2 T_3 ... T_N
$I_1 \cdot T_1$ $I_1 \cdot T_2$ $I_1 \cdot T_3$ ... $I_1 \cdot T_N$
A photo of a dog.

Multimodal Retrieval



Retrieval Pipeline Example


```
# 1. Define input and output modalities

cond_domains = ['caption']
target_domains = ['tok_dinov2_global']
tokens_per_target = [16]
autoregression_schemes = ['roar']
decoding_steps = [1]
token_decoding_schedules = [ 'linear' ]
temps = [0.1]
temp_schedules = ['constant']
cfg_scales = [1.0]
cfg_schedules = ['constant']
cfg_grow_conditioning = True
top_p, top_k = 0.8, 0.0

schedule = build_chained_generation_schedules(
    cond_domains=cond_domains, target_domains=target_domains,
    tokens_per_target=tokens_per_target, autoregression_schemes=autoregression_schemes,
    decoding_steps=decoding_steps, token_decoding_schedules=token_decoding_schedules,
    temps=temps, temp_schedules=temp_schedules, cfg_scales=cfg_scales,
    cfg_schedules=cfg_schedules, cfg_grow_conditioning=cfg_grow_conditioning,
)
```

Retrieval Pipeline Example


```
# 2. Create your input data

caption = "A cat sitting on a couch"
batched_sample = {}

# Initialize target modalities
for target_mod, ntoks in zip(target_domains, tokens_per_target):
    batched_sample = init_empty_target_modality(
        batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device
    )

# Initialize input modalities
batched_sample = custom_text(
    batched_sample, input_text=caption, text_tokenizer=text_tok,
    eos_token='[EOS]', key='caption', device=device, target_max_len=256,
    start_token='[S_1]'
)
```

Retrieval Pipeline Example

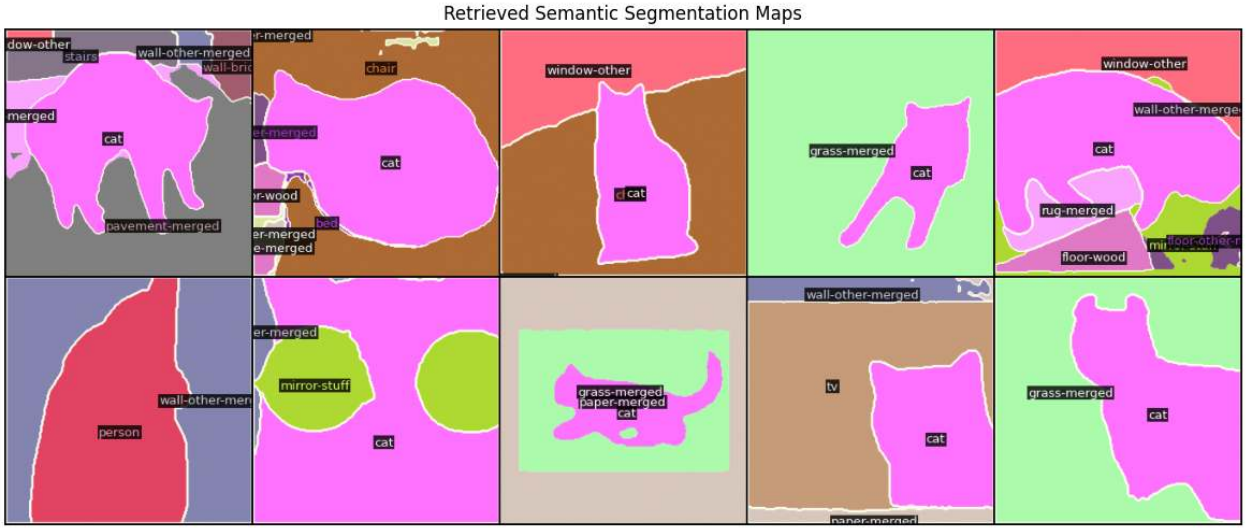

```
# 3. Get the model prediction

out_dict = sampler.generate(
    batched_sample, schedule, text_tokenizer=text_tok,
    verbose=True, seed=0,
    top_p=top_p, top_k=top_k,
)
dec_dict = decode_dict(
    out_dict, toks, text_tok,
    image_size=224, patch_size=16,
    decoding_steps=50
)
```

Retrieval Pipeline Example


```
# 4. Do the retrieval and display
index = faiss.IndexFlatL2(semseg_retrieval_set_embeddings.size(1))
index.add(semseg_retrieval_set_embeddings.cpu().numpy())

query_feature = dec_dict['tok_dinov2_global']
query_feature = query_feature.unsqueeze(0).unsqueeze(2).unsqueeze(2).cpu().numpy()

k =  10
distances, indices = index.search(query_feature.reshape(1, -1), k)

# Display the query image
plt.figure(figsize=(20,6))
gs = gridspec.GridSpec(1, 2, width_ratios=[1, 5])
ax0 = plt.subplot(gs[0])
metadata_text = ',\n'.join([f'{k}: {v:.2f}' if isinstance(v, float) else f'{k}: {v}' for k, v in
metadata_dict.items()])
plot_text_in_square(ax0, "Query input: " + caption + ", " + metadata_text, wrap_width=18, fontsize=14)
ax0.axis('off')

# Display the retrieved images
retrieved_images = []
for i in indices[0]:
    batched_sample = semseg_retrieval_set_tokens[i]
    retrieved_images.append(decode_tok_semseg(np.ones((224, 224, 3)), batched_sample, toks, 'tok_semseg@224'))

denormalized_retrieved_images = [torch.from_numpy(img).permute(2,0,1) for img in retrieved_images]
retrieved_grid = make_grid(denormalized_retrieved_images, nrow=5)
retrieved_grid = retrieved_grid.permute(1, 2, 0)

ax1 = plt.subplot(gs[1])
ax1.imshow(retrieved_grid)
ax1.set_title("Retrieved Semantic Segmentation Maps")
ax1.axis('off')
plt.show()
```

Retrieval Results

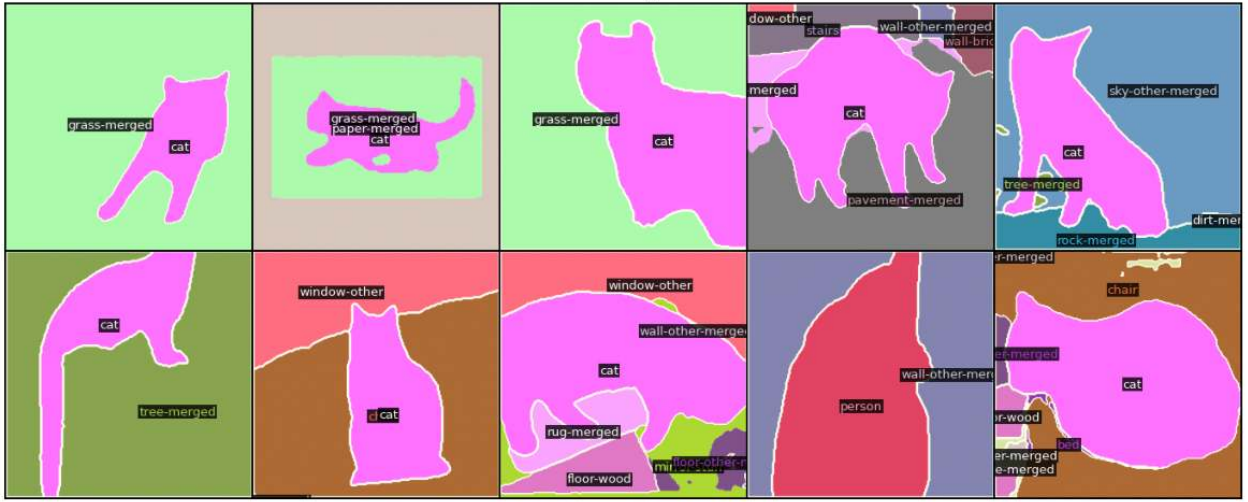
Query input: A cat sitting on a couch, semantic_diversity: 20



Retrieval Results

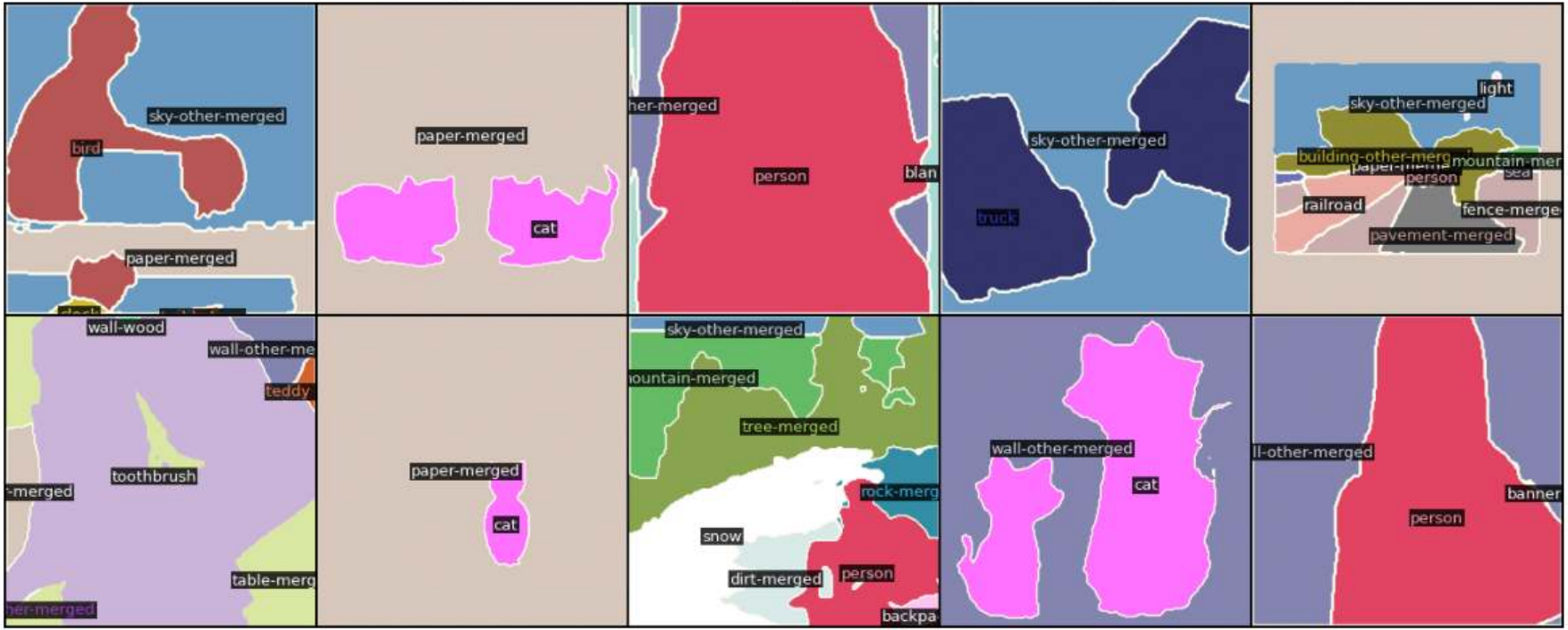
Query Image

Retrieved Semantic Segmentation Maps



ImageBind Retrieval

Task: Use alternative deep features to find semantic segmentation maps containing cats in the 10 examples.



ImageBind Retrieval


```
# 1. load the semseg and imagebind tokenizers

toks = {
    'tok_semseg': VQVAE.from_pretrained(
        'EPFL-VILAB/4M_tokenizers_semseg_4k_224-448'
    ).eval().to(device),
    'tok_dinov2_global': VQVAE.from_pretrained(
        'EPFL-VILAB/4M_tokenizers_DINOv2-B14-global_8k_16_224'
    ).eval().to(device),
    'tok_imagebind_global': VQVAE.from_pretrained(
        'EPFL-VILAB/4M_tokenizers_ImageBind-H14-global_8k_16_224'
    ).eval().to(device)
}
text_tok = Tokenizer.from_file('toks/text_tokenizer_4m_wordpiece_30k.json')
```

ImageBind Retrieval


```
# 2. Iterate through the 10 segmentation images and extract their ImageBind features.

cond_domains = ['tok_semseg@224']
target_domains = ['tok_imagebind_global']
tokens_per_target = [16]
autoregression_schemes = ['roar']
decoding_steps = [1]
token_decoding_schedules = ['linear']
temps = [0.1]
temp_schedules = ['constant']
cfg_scales = [1.0]
cfg_schedules = ['constant']
cfg_grow_conditioning = True
top_p, top_k = 0.8, 0.0

schedule = build_chained_generation_schedules(
    cond_domains=cond_domains, target_domains=target_domains,
    tokens_per_target=tokens_per_target, autoregression_schemes=autoregression_schemes,
    decoding_steps=decoding_steps, token_decoding_schedules=token_decoding_schedules,
    temps=temps, temp_schedules=temp_schedules, cfg_scales=cfg_scales,
    cfg_schedules=cfg_schedules, cfg_grow_conditioning=cfg_grow_conditioning,
)
```

ImageBind Retrieval


```
semseg_retrieval_set_embeddings_dino = []
for token in semseg_retrieval_subset_tokens:
    batched_sample = { # Make a batched sample for each output
        'tok_semseg@224': {
            'tensor': token['tok_semseg@224']['tensor'], # Batched tensor
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_target):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domains:
        batched_sample = init_full_input_modality(
            batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]")
        )

    # Get the model prediction
    out_dict = sampler.generate(
        batched_sample, schedule, text_tokenizer=text_tok, verbose=True, seed=0, top_p=top_p, top_k=top_k,
    )
    dec_dict = decode_dict(
        out_dict, toks, text_tok, image_size=224, patch_size=16, decoding_steps=50
    )

    semseg_retrieval_set_embeddings_dino.append(dec_dict['tok_imagebind_global'].unsqueeze(0).to(dtype=torch.float32))

semseg_retrieval_set_embeddings_dino = torch.cat(semseg_retrieval_set_embeddings_dino)

index = faiss.IndexFlatL2(semseg_retrieval_set_embeddings_dino.size(1))
index.add(semseg_retrieval_set_embeddings_dino.cpu().numpy())
```

ImageBind Retrieval


```
# 3. Create the query input and extract the corresponding ImageBind features.

cond_domains = ['caption', 'metadata']
target_domains = ['tok_imagebind_global']
tokens_per_target = [16]
autoregression_schemes = ['roar']
decoding_steps = [1]
token_decoding_schedules = [ 'linear' ]
temps = [0.1]
temp_schedules = ['constant']
cfg_scales = [1.0]
cfg_schedules = ['constant']
cfg_grow_conditioning = True
top_p, top_k = 0.8, 0.0

schedule = build_chained_generation_schedules(
    cond_domains=cond_domains, target_domains=target_domains,
    tokens_per_target=tokens_per_target, autoregression_schemes=autoregression_schemes,
    decoding_steps=decoding_steps, token_decoding_schedules=token_decoding_schedules,
    temps=temps, temp_schedules=temp_schedules, cfg_scales=cfg_scales,
    cfg_schedules=cfg_schedules, cfg_grow_conditioning=cfg_grow_conditioning,
)
```

ImageBind Retrieval


```
semseg_retrieval_set_embeddings_dino = []
for token in semseg_retrieval_subset_tokens:
    batched_sample = { # Make a batched sample for each output
        'tok_semseg@224': {
            'tensor': token['tok_semseg@224']['tensor'], # Batched tensor
            'input_mask': torch.zeros(1, 196, dtype=torch.bool, device=device), # False = used as input, True = ignored
            'target_mask': torch.ones(1, 196, dtype=torch.bool, device=device), # False = predicted as target, True = ignored
        }
    }

    # Initialize target modalities
    for target_mod, ntoks in zip(target_domains, tokens_per_target):
        batched_sample = init_empty_target_modality(batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device)

    # Initialize input modalities
    for cond_mod in cond_domains:
        batched_sample = init_full_input_modality(
            batched_sample, MODALITY_INFO, cond_mod, device, eos_id=text_tok.token_to_id("[EOS]")
        )

    # Get the model prediction
    out_dict = sampler.generate(
        batched_sample, schedule, text_tokenizer=text_tok, verbose=True, seed=0, top_p=top_p, top_k=top_k,
    )
    dec_dict = decode_dict(
        out_dict, toks, text_tok, image_size=224, patch_size=16, decoding_steps=50
    )

    semseg_retrieval_set_embeddings_dino.append(dec_dict['tok_imagebind_global'].unsqueeze(0).to(dtype=torch.float32))

semseg_retrieval_set_embeddings_dino = torch.cat(semseg_retrieval_set_embeddings_dino)

index = faiss.IndexFlatL2(semseg_retrieval_set_embeddings_dino.size(1))
index.add(semseg_retrieval_set_embeddings_dino.cpu().numpy())
```

ImageBind Retrieval

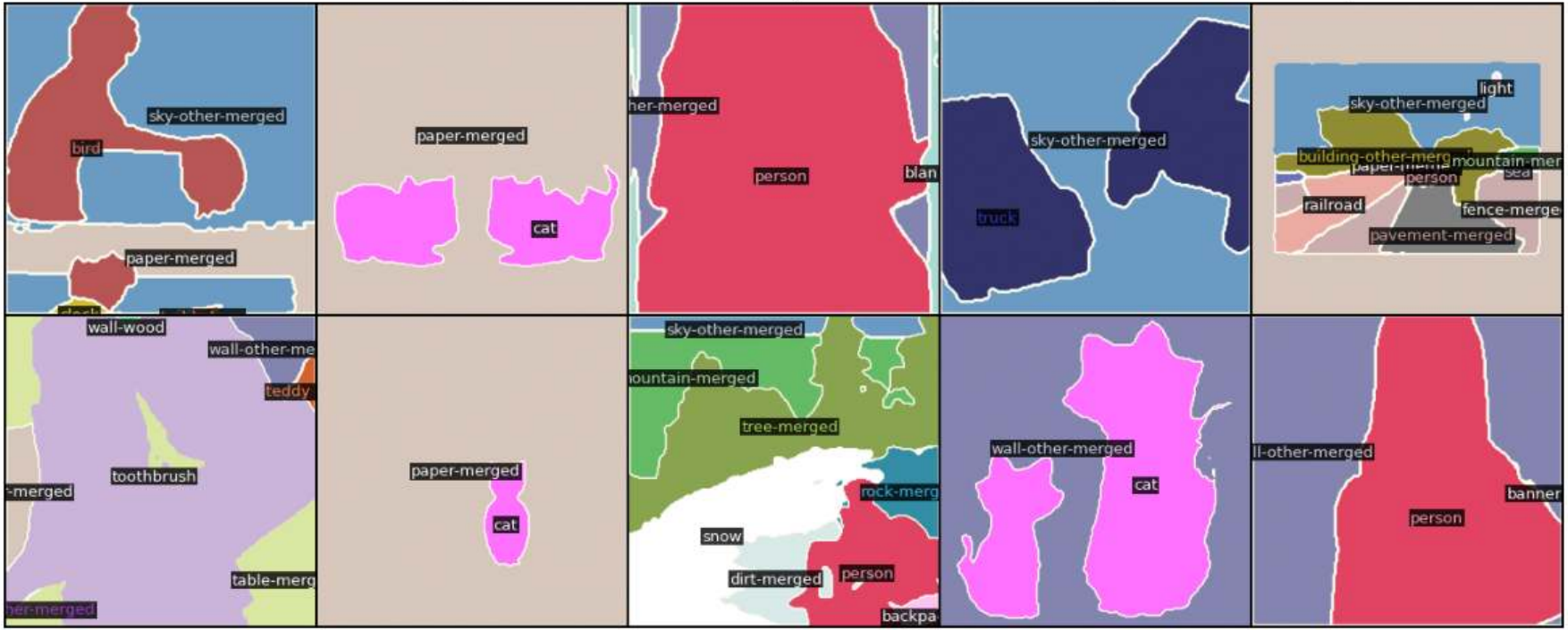

```
# Query with a caption and metadata

caption = 'Cat' # Thing should be a cat

metadata_transform = MetadataTransform(shuffle=False, random_trunc=False,
return_chunks=False)
metadata_dict = {'semantic_diversity': 1} # Only thing in the picture
metadata_str = metadata_transform.metadata_to_string(metadata_dict) + ' [S_1]'

batched_sample = {}
# Initialize target modalities
for target_mod, ntoks in zip(target_domains, tokens_per_target):
    batched_sample = init_empty_target_modality(
        batched_sample, MODALITY_INFO, target_mod, 1, ntoks, device
    )

batched_sample = custom_text( # Add caption
    batched_sample, input_text=caption, text_tokenizer=text_tok, eos_token='[EOS]',
    key='caption', device=device, target_max_len=256, start_token='[S_1]'
)
batched_sample = custom_text( # Add metadata
    batched_sample, input_text=metadata_str, text_tokenizer=text_tok,
    eos_token='[EOS]', key='metadata', device=device
)
```

ImageBind Retrieval


```python
# 3. Get the model prediction

out_dict = sampler.generate(
    batched_sample, schedule, text_tokenizer=text_tok,
    verbose=True, seed=0,
    top_p=top_p, top_k=top_k,
)
dec_dict = decode_dict(
    out_dict, toks, text_tok,
    image_size=224, patch_size=16,
    decoding_steps=50
)

query_feature = dec_dict['tok_imagebind_global']
query_feature = query_feature.unsqueeze(0).unsqueeze(2).unsqueeze(2).cpu().numpy()

# Query for the 3 most similar images in the subset using ImageBind features
k =  3
distances, indices = index.search(query_feature.reshape(1, -1), k)

# Get images
retrieved_images = []
for i in indices[0]:
    batched_sample = semseg_retrieval_subset_tokens[i]
    retrieved_images.append(
        decode_tok_semseg(np.ones((224, 224, 3)), batched_sample, toks, 'tok_semseg@224')
    )
```

ImageBind Retrieval

Task: Use alternative deep features to find semantic segmentation maps containing cats in the 10 examples.



ImageBind Retrieval

Query input: Cat