Dataloader:

- Create a file tumor_dataset.py and put under MiniGPT-4/minigpt4/datasets/datasets/tumor_dataset_concat.py
- Used to let the llm know the correspondence of jpegs and the explanation of it in the summary file.
- We need to define our vis processor, text processor here.
- Data:
    - **vis_root_dict**: the directory containing the jpegs.
    - **jsonl_files_dict**: the directory containing the json files.
    - **transform**: we can transform the photo to make it standardized so that the llm can perform better.

We extract the following:

- **images:** the actual image tensor for 4 MRI modalities
- **instruction_input:** same for every picture. The prompt that we talked about before.
- **answer:** our answer of coordinates as a string
- **bbox:** the actual coordinates

```
import json
import os
from torch.utils.data import Dataset
from PIL import Image
import torch
from torchvision import transforms

# Define the transformations
transform = transforms.Compose([
   transforms.Resize((224, 224)),
   transforms.ToTensor(),
   transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

class TumorDatasetConcat(Dataset):
   def __init__(self, vis_processor, text_processor, vis_root_dict,
jsonl_files_dict, transform=None):
       self.vis_processor = vis_processor
       self.text_processor = text_processor
       self.vis_root_dict = vis_root_dict
       self.jsonl_files_dict = jsonl_files_dict
       self.data = self._load_data()
       self.transform = transform

   def _load_data(self):
       data = []
       for tumor_type, jsonl_file in self.jsonl_files_dict.items():
```

```python
            vis_root = self.vis_root_dict[tumor_type]
            with open(jsonl_file, 'r') as f:
                for line in f:
                    sample = json.loads(line)
                    # Construct full paths for each modality
                    for key in sample.keys():
                        if key.endswith('_path'):
                            sample[key] = os.path.join(vis_root, sample[key])
                    # Parse the bounding box coordinates from the answer field
                    bbox_str = sample['answer'].split('<box>')[0].strip()
                    bbox = json.loads(bbox_str)
                    sample['bbox'] = bbox
                    data.append(sample)
        return data

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

    def __getitem__(self, idx):
        sample = self.data[idx]

        # Load images for each modality
        image_paths = [sample[key] for key in sample.keys() if
key.endswith('_path')]
        images = [Image.open(image_path).convert('RGB') for image_path in
image_paths]

        if self.transform:
            images = [self.transform(image) for image in images]

        # Stack images to create a tensor of shape (num_modalities, C, H, W)
        images = torch.stack(images)

        instruction =
"<Img><ImageHere></Img><Img><ImageHere></Img><Img><ImageHere></Img><Img><ImageHere>
</Img> Where is the tumor? <box>"
        answer = sample['answer']

        return {
            'images': images,
            'instruction_input': instruction,
            'answer': answer,
            'bbox': torch.tensor(sample['bbox'], dtype=torch.float32)
        }
```

Training config file

- Create a new config file MiniGPT-4/train_configs/tumor_segmentation_finetune.yaml.
- The file deviates from the standard finetuning file MiniGPT-4/train_configs/minigpt4_stage2_finetune.yaml.
- This file defines the standard things in dataloader, such as where your last checking point's ckpt is, what your vis_processor and text_processor is.
- The architecture should be changed to minigpt4_concat, which we will create below

```yaml
model:
 arch: minigpt4_concat
 model_type: pretrain_vicuna0

 max_txt_len: 160
 end_sym: "###"
 prompt_path: "prompts/alignment.txt"
 prompt_template: '###Human: {} ###Assistant: '
 ckpt: '/home/Intern1/Yun_SRP/MiniGPT-4/vicunav0-7b/prerained_minigpt4_7b.pth'

datasets:
 tumor:
   batch_size: 1
   vis_processor:
     train:
       name: "blip2_image_train"
       image_size: 224
   text_processor:
     train:
       name: "blip_caption"

run:
 task: image_text_pretrain
 # optimizer
 lr_sched: "linear_warmup_cosine_lr"
 init_lr: 3e-5
 min_lr: 1e-5
 warmup_lr: 1e-6

 weight_decay: 0.05
 max_epoch: 100
 iters_per_epoch: 5000
 num_workers: 4
 warmup_steps: 200

 seed: 42
 output_dir: "output/minigpt4_stage2_finetune"

 amp: True
```

```yaml
resume_ckpt_path: null

evaluate: False
train_splits: ["train"]

device: "cuda"
world_size: 1
dist_url: "env://"
distributed: True

wandb_log: True
job_name: minigpt4_finetune
```

Config file for dataloader

- Create a folder to store your dataset, where I named it as brats23 at MiniGPT-4/minigpt4/configs/datasets/brats23_concat/.
- Create defaults.yaml and store it under brats23_concat.
- This a specialized file that tells the llm where is your root for photo and jsonl, and anything nonstandard that your created at the dataloader.
- The name under datasets: should be the same as the one you used to register the dataloader.

```yaml
datasets:
 tumor_concat:
   data_type: images

   build_info:
     vis_root:
       GLI: /home/Data1/Brain_Tumor_data/BraTS23/newGLI
       MEN: /home/Data1/Brain_Tumor_data/BraTS23/newMEN
       MET: /home/Data1/Brain_Tumor_data/BraTS23/newMET

     jsonl_files:
       GLI: /home/Intern1/Yun_SRP/MiniGPT-4/brats23/train_concat/summary_GLI.jsonl
       MEN: /home/Intern1/Yun_SRP/MiniGPT-4/brats23/train_concat/summary_MEN.jsonl
       MET: /home/Intern1/Yun_SRP/MiniGPT-4/brats23/train_concat/summary_MET.jsonl

   transform:
     train:
       - type: Resize
         size: [224, 224]
       - type: Normalize
         mean: [0.485, 0.456, 0.406]
         std: [0.229, 0.224, 0.225]
```

Register data loader

- Edit the file MiniGPT-4/minigpt4/datasets/builders/image_text_pair_builder.py.
- You need to let your llm know about your dataloader.
- The name inside @registry.register_builder('xxx') should be the same as the name in your config file for data loader
- Write about the basic info about your dataloader, deviating from the examples shown below.
    - The things inside DATASET_CONFIG_DICT should be the path to your default.yaml wrote above.

```
from minigpt4.datasets.datasets.tumor_dataset_concat import TumorDatasetConcat

@registry.register_builder('tumor_concat')
class TumorDatasetBuilderConcat(BaseDatasetBuilder):
   train_dataset_cls = TumorDataset
   DATASET_CONFIG_DICT = {
       "default": "configs/datasets/brats23_concat/defaults.yaml",
   }

   def build_datasets(self):
       logging.info("Building datasets...")
       self.build_processors()
       build_info = self.config.build_info
       datasets = dict()

       transform = transforms.Compose([
           transforms.Resize((224, 224)),
           transforms.ToTensor(),
           transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225]),
       ])

       # create datasets
       dataset_cls = self.train_dataset_cls
       datasets['train'] = dataset_cls(
           vis_processor=self.vis_processors["train"],
           text_processor=self.text_processors["train"],
           vis_root_dict=dict(build_info.vis_root),
           jsonl_files_dict=dict(build_info.jsonl_files),
           transform=transform
       )
       return datasets
```

New minigpt4.py

- Create a new MiniGPT-4/minigpt4/models/minigpt4_concatenation.py
- Change its name to

```
class MiniGPTBaseConcat(BaseModel):
```

- Change how we encode the image

```
def encode_img(self, images):
        device = images[0].device

        # Ensures each image tensor has the correct shape (batch_size, channels,
height, width)
        if len(images[0].shape) > 4:
            images = [img.reshape(-1, *img.shape[-3:]) for img in images]

        # Process each modality separately and concatenate their embeddings
        image_embeds_list = []
        for img in images:
            # Use self.maybe_autocast() to enable mixed precision if supported
            with self.maybe_autocast():
                # Encode the image using the visual encoder and normalize the vision
features
                image_embeds = self.ln_vision(self.visual_encoder(img)).to(device)
                image_embeds_list.append(image_embeds)

        # Concatenate the embeddings from different modalities along the feature
dimension
        image_embeds = torch.cat(image_embeds_list, dim=1)

        if self.has_qformer:
            image_atts = torch.ones(image_embeds.size()[:-1],
dtype=torch.long).to(device)
            query_tokens = self.query_tokens.expand(image_embeds.shape[0], -1, -1)
            query_output = self.Qformer.bert(
                query_embeds=query_tokens,
                encoder_hidden_states=image_embeds,
                encoder_attention_mask=image_atts,
                return_dict=True,
            )
            inputs_llama = self.llama_proj(query_output.last_hidden_state)
        else:
            image_embeds = image_embeds[:, 1:, :]
            bs, pn, hs = image_embeds.shape
            image_embeds = image_embeds.view(bs, int(pn / len(images)), int(hs *
len(images)))
            inputs_llama = self.llama_proj(image_embeds)

        # Create an attention mask for the LLaMA model
        atts_llama = torch.ones(inputs_llama.size()[:-1],
dtype=torch.long).to(device)
```

```
        return inputs_llama, atts_llama
```

Mew minigpt_base.py

- Create a new MiniGPT-4/minigpt4/models/minigpt_base_concatenation.py
- Change the registration to the following

```
@registry.register_model("minigpt4_concat")
class MiniGPT4Concat(MiniGPTBase):
```

Changing def __init__(:

Change lora_r = 8 to enable LoRa:

```
def __init__(
        self,
        vit_model="eva_clip_g",
        img_size=224,
        drop_path_rate=0,
        use_grad_checkpoint=False,
        vit_precision="fp16",
        freeze_vit=True,
        llama_model="",
        max_txt_len=32,
        max_context_len=3800,
        prompt_template="",
        end_sym='\n',
        low_resource=False,  # use 8 bit and put vit in cpu
        device_8bit=0,  # the device of 8bit model should be set when loading and
cannot be changed anymore.
        lora_r=8,  # set lora_r to a value other than 0 to use LoRA
        lora_target_modules=["q_proj", "v_proj"],
        lora_alpha=16,
        lora_dropout=0.05,
    ):
```

For saving memory:

- Change everything in “cpu” to “device” with

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Add special tokens: let your model understand <box>

```
        BOX_TOKEN = '<box>'
        self.llama_tokenizer.add_tokens([BOX_TOKEN], special_tokens=True)
        self.llama_model.resize_token_embeddings(len(self.llama_tokenizer))
```

Add box decoder: used to decode the embeddings from llm to meaningful coordinates. Also initialize its weight

- We will normalize the bounding box, hence we add a sigmoid() to ensure the predicted bounding box is always between 0 and 1.

```
        config = self.llama_model.config
```

```
        self.loc_decoder = nn.Sequential(
            nn.Linear(config.hidden_size, config.hidden_size // 2),
            nn.ReLU(),
            nn.Linear(config.hidden_size // 2, 4),
            nn.Sigmoid()
        )
        self.loc_decoder.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                nn.init.constant_(module.bias, 0)
```

Change get_context_emb:

- The prompt will be segmented and convert to
  <Img> → emb1, <Img> → emb2, <Img> → emb3, <Img> → emb4, Where is the tumor? <box> → emb5
- Then intervene with the image embedding to create
  mixed_embs = [emb1, img_emb1, emb2, img_emb2, emb3, img_emb3, emb4, img_emb4, emb5]

```
def get_context_emb(self, prompt, img_list):
        device = img_list[0].device

        # Split the prompt at '<ImageHere>' placeholders
        prompt_segs = prompt.split('<ImageHere>')
        assert len(prompt_segs) == len(img_list) + 1, "Unmatched numbers of image placeholders and images."

        seg_tokens = []

        # Tokenize the text segments
        for i, seg in enumerate(prompt_segs):
            if i == 0:
                seg_tokens.append(self.llama_tokenizer(seg, return_tensors="pt", add_special_tokens=True).to(device).input_ids)
            else:
                seg_tokens.append(self.llama_tokenizer('<Img>' + seg, return_tensors="pt", add_special_tokens=True).to(device).input_ids)

        # Embed the tokenized text segments
        seg_embs = [self.embed_tokens(seg_t) for seg_t in seg_tokens]

        mixed_embs = []
```

```
        # Interleave the embeddings of the text segments and images
        for i, img_emb in enumerate(img_list):
            mixed_embs.append(seg_embs[i])
            mixed_embs.append(img_emb)
        mixed_embs.append(seg_embs[-1])

        # Concatenate all embeddings
        mixed_embs = torch.cat(mixed_embs, dim=1)
        return mixed_embs
```

Change preparing_embedding:

- Change how we read and encode img:

```
if 'images' in samples:
            img_embeds, img_atts = self.encode_img(samples["images"])
        else:
            img_embeds = img_atts = None
```

- Let the model read the location targets.

```
loc_targets = torch.tensor(samples["bbox"], dtype=torch.float32).to(self.device)
```

```
return cond_embeds, cond_atts, regress_embeds, regress_atts, part_targets,
loc_targets, regress_token_ids
```

Change forward:

- Read the loc_targets

```
cond_embeds, cond_atts, regress_embeds, regress_atts, part_targets, loc_targets,
regress_token_ids = self.preparing_embedding(samples)
```

- Return the giou loss + f1loss for bounding box if <box> is shown:

```
# Bounding Box Prediction
        if loc_targets is not None:
            loc_positions = (regress_token_ids ==
self.llama_tokenizer.convert_tokens_to_ids('<box>')).nonzero(as_tuple=True)
            selected_hidden_states = outputs.hidden_states[-1][loc_positions]
            #shape of selected_hidden_states (1,4096)
            pred_locs = self.loc_decoder(selected_hidden_states)

            if pred_locs.dim() == 1:
                pred_locs = pred_locs.unsqueeze(0)
            if loc_targets.dim() == 1:
                loc_targets = loc_targets.unsqueeze(0)

            loc_targets_normalized = loc_targets.clone()
            loc_targets_normalized[:, 0::2] /= 224  # assuming image width is 224
            loc_targets_normalized[:, 1::2] /= 224  # assuming image height is 224

            print(f"Predicted Locations: {pred_locs}")
```

```
        print(f"Target Locations: {loc_targets}")
        print(f"Target Locations: {loc_targets_normalized}")

        # Calculate IoU loss
        giou_loss = self.calculate_iou_loss(pred_locs, loc_targets_normalized)

        l1_loss = torch.nn.functional.l1_loss(pred_locs, loc_targets_normalized)

        total_loss = loss + giou_loss + l1_loss
        print(f"giou_loss: {giou_loss}")
        print(f"l1_loss: {l1_loss}")
        print(f"Total Loss: {total_loss}")
    else:
        total_loss = loss

    return {"loss": total_loss}
```

Calculate iou_loss:

- Need to change the coordinate to (minx, miny, maxx, maxy) format
- Calculate the intersection box area and enclose box area, calculate the giou which punishes for box being too far.
- Add some epsilon to ensure we do not handel nan when the intersection of predicted and actual bounding box is 0, which will cause division by 0.

```
import torch

def box_area(boxes):
   return (boxes[:, 2] - boxes[:, 0]) * (boxes[:, 3] - boxes[:, 1])

def generalized_box_iou(boxes1, boxes2, epsilon=1e-7):
   # Ensure that the coordinates are in the correct format
   boxes1 = boxes1.clone()
   boxes2 = boxes2.clone()

   # Swap coordinates to make sure they are in (minx, miny, maxx, maxy) format
   boxes1[:, [0, 2]] = torch.sort(boxes1[:, [0, 2]], dim=1)[0]
   boxes1[:, [1, 3]] = torch.sort(boxes1[:, [1, 3]], dim=1)[0]
   boxes2[:, [0, 2]] = torch.sort(boxes2[:, [0, 2]], dim=1)[0]
   boxes2[:, [1, 3]] = torch.sort(boxes2[:, [1, 3]], dim=1)[0]

   # Intersection box coordinates
   #left bottom
   lb = torch.max(boxes1[:, None, :2], boxes2[:, :2])  # [N,M,2]
   #right top
   rt = torch.min(boxes1[:, None, 2:], boxes2[:, 2:])  # [N,M,2]
```

```python
    # Width and height of intersection area
    wh = (rt - lb).clamp(min=0)  # [N,M,2]

    # Intersection area
    inter = wh[:, :, 0] * wh[:, :, 1]  # [N,M]

    area1 = box_area(boxes1)  # [N]
    area2 = box_area(boxes2)  # [M]

    union = area1[:, None] + area2 - inter  # [N,M]
    iou = inter / (union + epsilon)  # [N,M]

    # Enclosing box coordinates (smallest box to enclose the prediction and
bounding)
    lb_enclosing = torch.min(boxes1[:, None, :2], boxes2[:, :2])
    rt_enclosing = torch.max(boxes1[:, None, 2:], boxes2[:, 2:])

    wh_enclosing = (rt_enclosing - lb_enclosing).clamp(min=0)  # [N,M,2]
    area_enclosing = wh_enclosing[:, :, 0] * wh_enclosing[:, :, 1]  # [N,M]

    giou = iou - (area_enclosing - union) / (area_enclosing + epsilon)

    # Clamp GIoU to the range [-1, 1]
    giou = torch.clamp(giou, min=-1.0, max=1.0)

    # Print detailed debug information
    print(f"Boxes1: {boxes1}")
    print(f"Boxes2: {boxes2}")
    print(f"lb: {lb}")
    print(f"rt: {rt}")
    print(f"wh: {wh}")
    print(f"inter: {inter}")
    print(f"iou: {iou}")
    print(f"union: {union}")
    print(f"lb_enclosing: {lb_enclosing}")
    print(f"rt_enclosing: {rt_enclosing}")
    print(f"wh_enclosing: {wh_enclosing}")
    print(f"area_enclosing: {area_enclosing}")
    print(f"giou: {giou}")

    return giou
```

```python
    def calculate_iou_loss(self, pred_boxes, target_boxes, image_size=(224, 224),
epsilon=1e-7):
        # Ensure the boxes have the right shape
```

```
        if pred_boxes.dim() == 1:
            pred_boxes = pred_boxes.unsqueeze(0)
        if target_boxes.dim() == 1:
            target_boxes = target_boxes.unsqueeze(0)

        pred_box_area = box_area(pred_boxes)
        target_box_area = box_area(target_boxes)

        # Calculate IoU directly
        iou = generalized_box_iou(pred_boxes, target_boxes, epsilon)

        # IoU loss
        iou_loss = 1 - iou.diagonal().mean()

        return iou_loss
```

Change multi_select:

- Select the best predicted bounding box.

```
for answer in answers:
            #Constructs a dictionary
            choice_samples = {
                'image': images,
                'instruction_input': texts,
                'answer': answer
            }
            # Calls the forward method to compute the loss for the current
choice_samples.
            output = self.forward(choice_samples, reduction='none')
            loss = output['loss'].reshape(-1, 1)
            pred_boxes = output.get('prgeneralized_box_ioued_boxes', None)

            all_losses.append(loss)
            if pred_boxes is not None:
                all_pred_boxes.append(pred_boxes)
```

Change generate:

- How the model will generate answer for a prompt after training.
- Let it also output hidden state

```
# Generation Process
        with self.maybe_autocast():
            outputs = self.llama_model.generate(
                inputs_embeds=embs,
                attention_mask=attn_mask,
                max_new_tokens=max_new_tokens,
                num_beams=num_beams,
```

```
                length_penalty=length_penalty,
                temperature=temperature,
                do_sample=do_sample,
                min_length=min_length,
                top_p=top_p,
                repetition_penalty=repetition_penalty,
                output_hidden_states=True,  # Ensure hidden states are included
                return_dict_in_generate=True,  # Ensure the output is a dictionary
            )
```

- If <box> present, use the decoder to decode the embedding of the hidden state.

```
        answers = []
        box_token_id = self.llama_tokenizer.convert_tokens_to_ids('<box>')

        for i, output_token in enumerate(outputs.sequences):
            output_texts = self.llama_tokenizer.decode(output_token,
skip_special_tokens=True)
            output_texts = output_texts.split('</s>')[0]  # remove the stop sign
</s>
            output_texts = output_texts.replace("<s>", "")
            output_texts = output_texts.split(r'[/INST]')[-1].strip()
            if '<box>' in texts[i]:
                import re
                bracket_content = re.findall(r'\[.*?\]', output_texts)
                if bracket_content:
                    # Extract the first match and convert it to a list of floats
                    coordinates = list(map(float,
bracket_content[0][1:-1].split(',')))
                if len(coordinates) == 4:  # Ensure there are exactly four
coordinates
                    answers.append(coordinates)
            else:
                answers.append(output_texts)
```

Change runner_base.py

- To let the model know that it should train the loc_decoder.
- Add p_loc_decoder and give it a different learning rate lr.

```
@property
    def optimizer(self):
        # TODO make optimizer class and configurations
        if self._optimizer is None:
            num_parameters = 0
            p_wd, p_non_wd, p_loc_decoder = [], [], []

            # Access the underlying model if it's wrapped in DDP
```

```python
            model_to_update = self.model.module if isinstance(self.model,
torch.nn.parallel.DistributedDataParallel) else self.model
            for n, p in self.model.named_parameters():
                if not p.requires_grad:
                    continue  # frozen weights
                if "loc_decoder" in n:
                    p_loc_decoder.append(p)
                elif p.ndim < 2 or "bias" in n or "ln" in n or "bn" in n:
                    p_non_wd.append(p)
                else:
                    p_wd.append(p)
                num_parameters += p.data.nelement()
            logging.info("number of trainable parameters: %d" % num_parameters)

            optim_params = [
                {
                    "params": p_wd,
                    "weight_decay": float(self.config.run_cfg.weight_decay),
                },
                {"params": p_non_wd, "weight_decay": 0},
                {"params": p_loc_decoder, "lr": 1e-3}
            ]

            beta2 = self.config.run_cfg.get("beta2", 0.999)
            self._optimizer = torch.optim.Adam(
                optim_params,
                lr=float(self.config.run_cfg.init_lr),
                weight_decay=float(self.config.run_cfg.weight_decay),
                betas=(0.9, beta2),
            )

        return self._optimizer
```

Register the model

- Change MiniGPT-4/minigpt4/models/__init__.py
- Add that

```python
from minigpt4.models.minigpt_base_concatenation import MiniGPTBaseConcat
from minigpt4.models.minigpt4_concatenation import MiniGPT4Concat
```

train

- Run the following code with your number of gpu.

‘’’

cd /home/Intern1/Yun_SRP/MiniGPT-4
conda activate minigptv

nohup bash -c 'CUDA_VISIBLE_DEVICES=1,2 python train.py --cfg-path /home/Intern1/Yun_SRP/MiniGPT-4/train_configs/tumor_segmentation_.yaml' > /home/Intern1/Yun_SRP/process_input_data/miniGPT4/logs_concat.txt 2>&1 &

disown
‘’’

- The output checkpoint will be saved at MiniGPT-4/minigpt4/output.
- The log during training is saved to /home/Intern1/Yun_SRP/process_input_data/miniGPT4/logs.txt
- It will not be disconnect if you are out of internet

Check if the process is running:
    ps -ef | grep train.py

Monitor the log file:
    tail -n 50 /home/Intern1/Yun_SRP/process_input_data/miniGPT4/logs_concat.txt

It will produce the following during training for each bounding box for debugging purpose:

Predicted Locations: tensor([[0.9707, 0.6221, 0.0373, 0.0054]], device='cuda:0', dtype=torch.float16, grad_fn=<SigmoidBackward0>)
Target Locations: tensor([[0.7946, 0.1964, 0.6071, 0.0446]], device='cuda:0')
Boxes1: tensor([[0.0373, 0.0054, 0.9707, 0.6221]], device='cuda:0', dtype=torch.float16, grad_fn=<CopySlices>)
Boxes2: tensor([[0.6071, 0.0446, 0.7946, 0.1964]], device='cuda:0')
lb: tensor([[[0.6071, 0.0446]]], device='cuda:0', grad_fn=<MaximumBackward0>)
rt: tensor([[[0.7946, 0.1964]]], device='cuda:0', grad_fn=<MinimumBackward0>)
wh: tensor([[[0.1875, 0.1518]]], device='cuda:0', grad_fn=<ClampBackward1>)
inter: tensor([[0.0285]], device='cuda:0', grad_fn=<MulBackward0>)
iou: tensor([[0.0494]], device='cuda:0', grad_fn=<DivBackward0>)
union: tensor([[0.5757]], device='cuda:0', grad_fn=<SubBackward0>)
lb_enclosing: tensor([[[0.0373, 0.0054]]], device='cuda:0', grad_fn=<MinimumBackward0>)
rt_enclosing: tensor([[[0.9707, 0.6221]]], device='cuda:0', grad_fn=<MaximumBackward0>)
wh_enclosing: tensor([[[0.9334, 0.6167]]], device='cuda:0', grad_fn=<ClampBackward1>)
area_enclosing: tensor([[0.5756]], device='cuda:0', grad_fn=<MulBackward0>)
giou: tensor([[0.0496]], device='cuda:0', grad_fn=<ClampBackward1>)
giou_loss: 0.9504191875457764
l1_loss: 0.3026945888996124
Total Loss: 3.9845523834228516

Eval

Write your own test.py like:

```
import json
import os
import torch
```

```python
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from PIL import Image
from torchvision import transforms
from tqdm import tqdm
import numpy as np
from minigpt4.common.config import Config
from minigpt4.common.eval_utils import init_model, eval_parser
from minigpt4.conversation.conversation import CONV_VISION_minigptv2
from minigpt4.utils.box_ops import box_area, generalized_box_iou

def list_of_str(arg):
   return list(map(str, arg.split(',')))

def load_image(image_path, transform, device):
   image = Image.open(image_path).convert("RGB")
   image = transform(image).unsqueeze(0).to(device)
   return image

def visualize_boxes(image, gt_box, pred_box):
   fig, ax = plt.subplots(1)
   ax.imshow(image)

   gt_box = gt_box.cpu().numpy().squeeze()
   pred_box = pred_box.cpu().numpy().squeeze()

   gt_rect = patches.Rectangle((gt_box[0], gt_box[1]), gt_box[2]-gt_box[0],
gt_box[3]-gt_box[1], linewidth=1, edgecolor='g', facecolor='none')
   pred_rect = patches.Rectangle((pred_box[0], pred_box[1]),
pred_box[2]-pred_box[0], pred_box[3]-pred_box[1], linewidth=1, edgecolor='r',
facecolor='none')

   ax.add_patch(gt_rect)
   ax.add_patch(pred_rect)

   plt.show()

def main():
   parser = eval_parser()
   parser.add_argument("--dataset", type=list_of_str, default='GLI,MEN,MET',
help="dataset to evaluate")
   args = parser.parse_args()
   cfg = Config(args)

   device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
    model, vis_processor = init_model(args)

    BOX_TOKEN = '<box>'
    model.llama_tokenizer.add_tokens([BOX_TOKEN], special_tokens=True)
    model.llama_model.resize_token_embeddings(len(model.llama_tokenizer))
    box_token_id = model.llama_tokenizer.convert_tokens_to_ids(BOX_TOKEN)

    conv_temp = CONV_VISION_minigptv2.copy()
    conv_temp.system = ""
    model.eval()

    test_sets = {
        "GLI": ("/home/Intern1/Yun_SRP/MiniGPT-4/brats23/test/summary_GLI.jsonl",
"/home/Data1/Brain_Tumor_data/BraTS23/newGLI"),
        "MEN": ("/home/Intern1/Yun_SRP/MiniGPT-4/brats23/test/summary_MEN.jsonl",
"/home/Data1/Brain_Tumor_data/BraTS23/newMEN"),
        "MET": ("/home/Intern1/Yun_SRP/MiniGPT-4/brats23/test/summary_MET.jsonl",
"/home/Data1/Brain_Tumor_data/BraTS23/newMET")
    }

    transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor()
    ])

    prompt_template = "<Img><ImageHere></Img> Where is the tumor? <box>"

    for test_name, (summary_path, image_dir) in test_sets.items():
        print(f"Testing on {test_name} set...")

        with open(summary_path, 'r') as file:
            test_data = [json.loads(line) for line in file]

        results = []
        total_giou = 0
        count = 0

        for item in tqdm(test_data):
            image_filename = item["figure"]
            image_path = os.path.join(image_dir, image_filename)
            if os.path.exists(image_path):
                image = load_image(image_path, transform, device)
                answers = model.generate(
                    images=image,
                    texts=[prompt_template]
```

```
                )
                ground_truth_box_str = item.get('answer', None)
                if ground_truth_box_str:
                    ground_truth_box_str =
ground_truth_box_str.split('<box>')[0].strip()
                    ground_truth_box = list(map(float,
ground_truth_box_str.strip('[]').split(',')))
                    ground_truth_box =
torch.tensor(ground_truth_box).unsqueeze(0).to(device)
                    predicted_box = torch.tensor(answers[0]).unsqueeze(0).to(device)
                    giou = generalized_box_iou(predicted_box, ground_truth_box)
                    total_giou += giou.item()
                    count += 1
                    print(f"Image: {image_filename}")
                    print(f"Ground Truth Box: {ground_truth_box}")
                    print(f"Predicted Box: {predicted_box}")
                results.append((image_filename, answers[0]))

        if count > 0:
            overall_giou = total_giou / count
        else:
            overall_giou = 0

        print(f'Overall gIoU for {test_name} set: {overall_giou * 100:.2f}/100')

if __name__ == "__main__":
    main()
```

Edit /home/Intern1/Yun_SRP/MiniGPT-4/eval_configs/minigpt4_eval.yaml

Run:
CUDA_VISIBLE_DEVICES=1 python /home/Intern1/Yun_SRP/MiniGPT-4/test.py --cfg-path /home/Intern1/Yun_SRP/MiniGPT-4/eval_configs/minigpt4_eval.yaml