Dataloader:

- Create a file tumor_dataset.py and put under MiniGPT-4/minigpt4/datasets/datasets/tumor_dataset_adv.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 TumorDatasetadv(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():
```

```
            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_adv, which we will create below

```yaml
model:
 arch: minigpt4_adv
 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_adv/.
- Create defaults.yaml and store it under brats23_adv.
- 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_adv:
   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_adv/summary_GLI.jsonl
       MEN: /home/Intern1/Yun_SRP/MiniGPT-4/brats23/train_adv/summary_MEN.jsonl
       MET: /home/Intern1/Yun_SRP/MiniGPT-4/brats23/train_adv/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_adv import TumorDatasetadv

@registry.register_builder('tumor_adv')
class TumorDatasetBuilderadv(BaseDatasetBuilder):
   train_dataset_cls = TumorDataset
   DATASET_CONFIG_DICT = {
       "default": "configs/datasets/brats23_adv/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_adv.py
- Change its name to

```
class MiniGPTBaseadv(BaseModel):
```

- Change how we encode the image

```
        return inputs_llama, atts_llama
```

New minigpt_base.py

- Create a new MiniGPT-4/minigpt4/models/minigpt_base_adv.py
- Change the registration to the following with the new MiniGPTBaseAdv used and imported.

```
from minigpt4.models.minigpt_base_adv import MiniGPTBaseadv

@registry.register_model("minigpt4_adv")
def encode_img(self, images):
        device = images[0].device if isinstance(images, list) else images.device

        if isinstance(images, torch.Tensor) and len(images.shape) == 5:
            print(f"Image input is a tensor. Number of images: {images.shape[1]}")
            images = torch.split(images, 1, dim=1)  # Split the tensor into a list
of tensors along the second dimension
            images = [img.squeeze(1) for img in images]  # Remove the singleton
dimension
        else:
            print("No image list find")

        if isinstance(images, list):
            if len(images[0].shape) > 4:
                images = [img.reshape(-1, *img.shape[-3:]) for img in images]

            # Process each modality separately and concatnate their embeddings
            image_features_list = []
            for img in images:
                # print(f"image shapes: {img.shape}")
                with self.maybe_autocast():
                    image_features = self.visual_encoder(img.to(device))
                    image_features_list.append(image_features)
                    # print(f"image_features 1: {image_features.shape}")

            # Concat the embeddings from different modalities along the feature
dimension
            common_features = torch.cat(image_features_list, dim=1)
            # print(f"common_features 1: {common_features.shape}")

            # Combine the embeddings into a single representation
            common_features= self.fusion_network(common_features)
            image_embeds = self.ln_vision(common_features)
            image_embeds=self.v_q_project(image_embeds)
```

```
            # print(f"image_embeds shape after v_q_project and reshape:
{image_embeds.shape}")

            image_embeds=image_embeds.view(image_embeds.size(0),1,-1)
            # print(f"image_embeds shape after view: {image_embeds.shape}")

            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,
                )
                if isinstance(query_output, dict):
                    inputs_llama =
self.llama_proj(query_output["last_hidden_state"])
                else:
                    inputs_llama = self.llama_proj(query_output[0])
            else:
                common_embeds = common_embeds[:, 1:, :]
                bs, pn, hs = common_embeds.shape
                common_embeds = common_embeds.view(bs, int(pn / len(images)), int(hs
* len(images)))
                inputs_llama = self.llama_proj(common_embeds)
            # Create an attention mask for the LLaMA model
            atts_llama = torch.ones(inputs_llama.size()[:-1],
dtype=torch.long).to(device)

        else:
            print("no image list find")
            image = images  # Single image case
            if len(image.shape) > 4:
                image = image.reshape(-1, *image.shape[-3:])

            with self.maybe_autocast():
                image_features = self.visual_encoder(image.to(device))
                image_embeds = self.ln_vision(image_features)
                image_embeds = self.v_q_project(image_embeds)
                # Reshape image to (batch_size, 1, remaining)
                image_embeds = image_embeds.view(image_embeds.size(0), 1, -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 / 4), int(hs * 4))
                inputs_llama = self.llama_proj(image_embeds)
            atts_llama = torch.ones(inputs_llama.size()[:-1],
dtype=torch.long).to(image.device)

        return inputs_llama, atts_llama
```

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,
    ):
```

<u>For saving memory:</u>

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

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

<u>Add special tokens:</u> 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))
```

<u>Add box decoder:</u> 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)
```

Add Fusion Network:

```
class FusionNetwork(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FusionNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
```

```
        x = self.fc2(x)
        return x
```

```
encoder_feature_num = self.visual_encoder.num_features # 768
        self.fusion_network = FusionNetwork(input_dim=encoder_feature_num * 4,
hidden_dim=2048, output_dim=encoder_feature_num)

        self.v_q_project = nn.Linear(encoder_feature_num, 1408)

        self.llama_proj = nn.Linear(
            img_f_dim, 4096
        )

        # Add Box Decoder
        config = self.llama_model.config
        self.bbox_decoder = nn.Sequential(
            nn.Linear(4096, 4096//2),
            nn.ReLU(),
            nn.Linear(4096//2, 4)
        )
```

Change forward:

```
if any('<box>' in ans for ans in samples['answer']):
            target_box = samples['bbox']
            target_box_tensor = torch.tensor([target_box], device='cuda')
            # print(f"target_box: {target_box}")

            box_positions = (targets == box_token_id).nonzero(as_tuple=True)
            print(f"box_positions{box_positions}")

            if box_positions[0].size(0) > 0:
                box_positions = box_positions[-1]
                #shape of selected_hidden_states (1,4096)
                location_state = hidden_states[box_positions].view(1, -1)
                print(f"shape of location state{location_state.shape}")
                pred_box = self.bbox_decoder(location_state)
                # Check if pred_box has the correct shape
                assert pred_box.shape == (1, 4), f"Predicted box shape is incorrect:
{pred_box.shape}"

                print(f"Predicted Locations: {pred_box}")
                print(f"Target Locations: {target_box_tensor}")

                # Calculate IoU loss
```

```
                # giou_loss = self.calculate_iou_loss(pred_box, target_box_tensor)
                giou_loss = calc_giou_loss(pred_box, target_box_tensor)

                l1_loss = torch.nn.functional.smooth_l1_loss(pred_box,
target_box_tensor)

                if giou_loss>0:
                    total_loss= loss + 0.2 * l1_loss + 1.2 * giou_loss
                else:
                    total_loss=0.8 * loss + 0.2 * l1_loss

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

        return {"loss": total_loss}
```

Register the model

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

```
from minigpt4.models.minigpt_base_adv import MiniGPTBaseadv
from minigpt4.models.minigpt4_adv import MiniGPT4Adv
```

train

- 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
- Run the following code with your number of gpu.
- The output checkpoint will be saved at MiniGPT-4/minigpt4/output.

‘’’

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

disown

‘’’

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_adv.txt

tail -n 50 -f /home/Intern1/Yun_SRP/process_input_data/miniGPT4/logs_adv.txt

Detach from the shell:

Press Ctrl+B followed by D.

Reattach from the shell:

tmux attach -t adv_training_session

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 os
import re
import json
import pprint
import argparse
import random
from collections import defaultdict
import numpy as np
```

```
from PIL import Image
from tqdm import tqdm
import torch
import torch.backends.cudnn as cudnn
import minigpt4.tasks as tasks
import torchvision.transforms as transforms
from torch.utils.data import Dataset, DataLoader
from minigpt4.processors import *
from torchvision.transforms.functional import InterpolationMode
from minigpt4.common.registry import registry
from minigpt4.processors.blip_processors import Blip2ImageTrainProcessor
from minigpt4.datasets.datasets.vqa_datasets import OKVQAEvalData, VizWizEvalData,
IconQAEvalData, GQAEvalData, VSREvalData, HMEvalData
from minigpt4.common.config import Config
from minigpt4.utils.box_ops import calc_iou, giou, denormalize_bounding_boxes

# Argument Parser

def parse_args():
   parser = argparse.ArgumentParser(description="Training")
   parser.add_argument("--cfg-path",
default='/home/Intern1/Yun_SRP/MiniGPT-4/eval_configs/minigpt4_eval.yaml',
                       help="path to configuration file.")
   parser.add_argument("--gpu-id", type=int, default=2, help="specify the gpu to
load the model.")
   parser.add_argument(
       "--options",
       nargs="+",
       help="override some settings in the used config, the key-value pair "
            "in xxx=yyy format will be merged into config file (deprecate), "
            "change to --cfg-options instead.",
   )
   args = parser.parse_args()
   return args

random.seed(42)
np.random.seed(42)
torch.manual_seed(42)

cudnn.benchmark = False
cudnn.deterministic = True

args = parse_args()
cfg = Config(args)
```

```python
device = torch.device('cuda')

task = tasks.setup_task(cfg)
datasets = task.build_datasets(cfg)

model_config = cfg.model_cfg
model_cls = registry.get_model_class(model_config.arch)
model = model_cls.from_config(model_config).to(device)

vis_processor_cfg = cfg.config['datasets']['tumor']['vis_processor']['train']
# vis_processor_cfg = cfg.config['datasets']['tumor_adv']['vis_processor']['train']


vis_processor =
registry.get_processor_class(vis_processor_cfg['name']).from_config(vis_processor_c
fg)

model = model.eval()

dataloader = DataLoader(datasets['tumor']['test'], batch_size=1)
# dataloader = DataLoader(datasets['tumor_adv']['test'], batch_size=1)

total_iou = 0
count = 0
for sample in dataloader:
   text = sample['instruction_input']
   answers, predict_boundingbox = model.generate(sample['image'], text)
   print(text)
   if predict_boundingbox != []:
       print(f"processing: {sample['image_path']}")

       if predict_boundingbox.dim() == 1:
           predict_boundingbox = predict_boundingbox.unsqueeze(0)

       gt_boxes_tensor =  torch.tensor([sample['bbox']], device='cuda')

       print(f"Predicted bounding box: {predict_boundingbox}")
       print(f"Answer: {gt_boxes_tensor}")

       if predict_boundingbox.dim() == 1:
           predict_boundingbox = predict_boundingbox.unsqueeze(0)
       if gt_boxes_tensor.dim() == 1:
           gt_boxes_tensor = gt_boxes_tensor.unsqueeze(0)

       iou = calc_iou(predict_boundingbox, gt_boxes_tensor)
```

```
        total_iou += iou.item()
        count += 1

        average_iou = total_iou / count if count > 0 else 0
        print(f'ith: {count}')
        print(f'Average IoU: {average_iou}')
```

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

Run:
CUDA_VISIBLE_DEVICES=3 python eval_seg.py --cfg-path /home/Intern1/Yun_SRP/MiniGPT-4/eval_configs/minigpt4_eval_adv.yaml > /home/Intern1/Yun_SRP/process_input_data/test_adv.txt 2>&1 &

tail -n 50 -f /home/Intern1/Yun_SRP/process_input_data/test_adv.txt