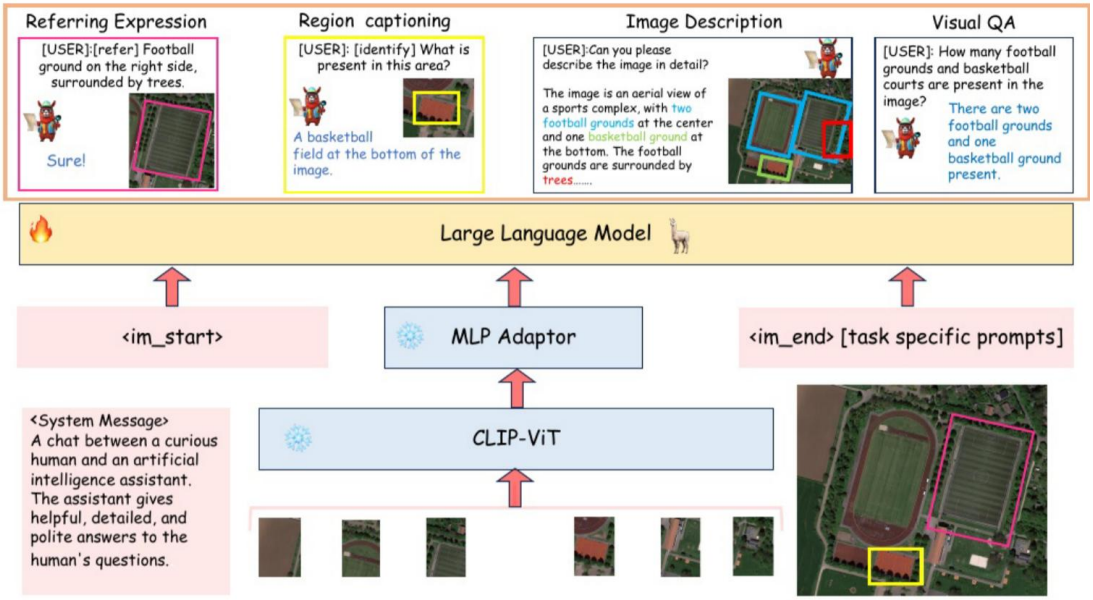
《GeoChat : Grounded Large Vision-Language Model for Remote Sensing》论文复现与改进

（武汉大学计算机学院）

1 算法模型简介

该论文发表在 CVPR2024 上
GeoChat:接地遥感 VLM



GeoChat 遵循 llva -v1.5 的架构，它由三个核心组件组成，全局图像编码器，MLP 适配器和 LLM。与 LLaVA 不同的是，添加了特定的任务提示符，用于指示模型所需的任务类型，即基础、图像级或区域级对话。

视觉主干。GeoChat 采用 CLIP-ViT(L-14)的预训练视觉主干，其输入分辨率为 336×336。这导致每张图像有效地有 576 个补丁。由于该分辨率不足以理解遥感图像中呈现的细节(例如，小物体和物体细节)，因此在基于变压器的 CLIP 模型中插入位置编码，以根据输入图像大小 504×504 进行缩放。虽然这导致补丁数量增加到几乎翻倍，但是这种增强的分辨率使其能够处理更大的图像尺寸，并且还支持高分辨率 RS 图像中更好的视觉接地。

MLP 跨模态适配器。用带有一个隐藏层的 MLP 适配器，将维度为 1024 的输出令牌投影到语言模型空间。该适配器的输入维数为 1024，输出大小为 4096 的向量，对应于 LLM 的输入大小。

大型语言模型。开源的 Vicuna-v1.5(7B)大型语言模型被用作 GeoChat 的基础。该语言模型在作者的框架中作为一个单一的界面，用于不同的视觉语言输入。为了完成不同的视觉语言任务，该模型直接依赖于 Vicuna-1.5(7B)语言库。

2 关键技术问题解释与分析

由于这个论文是遥感领域多模态大模型领域的，因此模型架构主要由三个关键部分有机组合而成。首先是视觉部分，它承担着接收并处理来自遥感影像的海量视觉信息的重任，诸如卫星拍摄的高清地面图像、航空摄影所捕捉到的地理风貌细节等，通过一系列复杂且精妙的卷积神经网络等技术，精准地提取其中的各类特征，为后续的分析提供坚实的数据基础。

然后着是文本部分，它能够对与遥感相关的文本描述进行深入剖析，这些文本主要是对图像的描述，借助自然语言处理中的词嵌入、句法分析等手段，将文字转化为模型能够理解的结构化信息，进而与视觉信息相互补充。

最后是图文对齐的连接头，它是连接视觉与文字这两个不同模态信息的关键桥梁，他的设计的架构与算法，致力于寻找视觉特征与文字特征之间的内在关联，使得两者能够在高维空间中实现精准对齐，确保模型在面对复杂的遥感任务时，能够充分融合多模态信息，从而给出更为准确、全面的分析结果，下面逐一讲解。

2.1 视觉部分

视觉部分采用的是 CLIP 编码器。

```python
class CLIPVisionTower(nn.Module):
    def clip_interpolate_embeddings(self, image_size=600, patch_size= 14):
        # Shape of pos_embedding is (1, seq_length, hidden_dim)
        state_dict = self.vision_tower.vision_model.embeddings.position_embedding.state_dict()
        pos_embedding = state_dict['weight']
        pos_embedding = pos_embedding.unsqueeze(0)
        n, seq_length, hidden_dim = pos_embedding.shape
        if n != 1:
            raise ValueError(f"Unexpected position embedding shape: {pos_embedding.shape}")

        new_seq_length = (image_size // patch_size) ** 2 + 1

        # Need to interpolate the weights for the position embedding.
        # We do this by reshaping the positions embeddings to a 2d grid, performing
        # an interpolation in the (h, w) space and then reshaping back to a 1d grid.
        if new_seq_length != seq_length:
            # The class token embedding shouldn't be interpolated so we split it up.
            seq_length -= 1
            new_seq_length -= 1
            pos_embedding_token = pos_embedding[:, :1, :]
            pos_embedding_img = pos_embedding[:, 1:, :]

            # (1, seq_length, hidden_dim) -> (1, hidden_dim, seq_length)
            pos_embedding_img = pos_embedding_img.permute(0, 2, 1)
            seq_length_1d = int(math.sqrt(seq_length))
            torch._assert(seq_length_1d * seq_length_1d == seq_length, "seq_length is not a perfect square!")

            # (1, hidden_dim, seq_length) -> (1, hidden_dim, seq_l_1d, seq_l_1d)
            pos_embedding_img = pos_embedding_img.reshape(1, hidden_dim, seq_length_1d, seq_length_1d)
            new_seq_length_1d = image_size // patch_size

            # Perform interpolation.
            # (1, hidden_dim, seq_l_1d, seq_l_1d) -> (1, hidden_dim, new_seq_l_1d, new_seq_l_1d)
            new_pos_embedding_img = nn.functional.interpolate(
                pos_embedding_img,
                size=new_seq_length_1d,
                mode='bicubic',
                align_corners=True,
            )

            # (1, hidden_dim, new_seq_l_1d, new_seq_l_1d) -> (1, hidden_dim, new_seq_length)
            new_pos_embedding_img = new_pos_embedding_img.reshape(1, hidden_dim, new_seq_length)

            # (1, hidden_dim, new_seq_length) -> (1, new_seq_length, hidden_dim)
            new_pos_embedding_img = new_pos_embedding_img.permute(0, 2, 1)
            new_pos_embedding = torch.cat([pos_embedding_token, new_pos_embedding_img], dim=1)[0]
            state_dict['weight'] = new_pos_embedding
            self.vision_tower.vision_model.embeddings.position_embedding = nn.Embedding(new_seq_length+1, hidden_dim)
            self.vision_tower.vision_model.embeddings.position_embedding.load_state_dict(state_dict)
            self.vision_tower.vision_model.embeddings.image_size = image_size
            self.vision_tower.vision_model.embeddings.patch_size = patch_size
            self.vision_tower.vision_model.embeddings.position_ids = torch.arange(new_seq_length+1).expand((1, -1))
```

代码定义了一个名为 CLIPVisionTower 的 PyTorch 神经网络模块，主要用于处理与 CLIP 相关的视觉任务。这个模块涉及对 CLIP 视觉模型的加载、特征提取、位置嵌入的插值处理以及对不同输入图像形式的处理。

首先看到 clip_interpolate_embeddings 函数，它的功能是对位置嵌入进行插值处理。首先从 self.vision_tower.vision_model.embeddings.position_embedding 中获取位置嵌入的状态字典，并提取其权重 pos_embedding 。然后计算新的序列长度 new_seq_length 。如果新的序列长度与原序列长度不同，将位置嵌入分为类标记嵌入和图像嵌入部分，对图像嵌入部分进行维度重排、形状重塑，以将其从 1D 序列形式转化为 2D 网格形式，然后使用双三次插值对其进行插值操作，将结果重新调整回 1D 序列形式，最后将类标记嵌入和插值后的图像嵌入拼接起来更新状态字典中的权重。此外，还更新了 self.vision_tower.vision_model.embeddings 的相关属性，包括 position_embedding、image_size、patch_size 和 position_ids。

```
def __init__(self, vision_tower, args, delay_load=False):
    super().__init__()

    self.is_loaded = False

    self.vision_tower_name = vision_tower
    self.select_layer = args.mm_vision_select_layer
    self.select_feature = getattr(args, 'mm_vision_select_feature', 'patch')

    if not delay_load:
        self.load_model()
    else:
        self.cfg_only = CLIPVisionConfig.from_pretrained(self.vision_tower_name)
        self.image_processor = CLIPImageProcessor.from_pretrained(self.vision_tower_name)
        self.vision_tower = CLIPVisionModel.from_pretrained(self.vision_tower_name)
        self.vision_tower.requires_grad_(False)
        self.clip_interpolate_embeddings(image_size=504, patch_size=14)
```

然后是__init__方法，它是类的构造函数。首先调用父类 nn.Module 的构造函数，设置 is_loaded 为 False，存储 vision_tower_name、select_layer 和 select_feature 信息。根据 delay_load 参数，决定是否立即加载模型。如果 delay_load 为 False，调用 load_model 函数加载模型；如果为 True，先仅从预训练中加载配置、图像处理器和视觉模型，设置 vision_tower 不进行梯度更新，并调用 clip_interpolate_embeddings 进行位置嵌入的插值操作。

```
def load_model(self):
    self.image_processor = CLIPImageProcessor.from_pretrained(self.vision_tower_name)
    self.vision_tower = CLIPVisionModel.from_pretrained(self.vision_tower_name)
    self.vision_tower.requires_grad_(False)
    self.clip_interpolate_embeddings(image_size=504, patch_size=14)

    self.is_loaded = True
    # print(self.is_loaded)

def feature_select(self, image_forward_outs):
    image_features = image_forward_outs.hidden_states[self.select_layer]
    if self.select_feature == 'patch':
        image_features = image_features[:, 1:]
    elif self.select_feature == 'cls_patch':
        image_features = image_features
    else:
        raise ValueError(f'Unexpected select feature: {self.select_feature}')
    return image_features
```

后面是 load_model 以及 feature_select，load_model 的功能是实际执行模型的

加载操作。从预训练中加载 CLIPImageProcessor 和 CLIPVisionModel，设置 vision_tower 不进行梯度更新，并调用 clip_interpolate_embeddings 进行位置嵌入的插值操作，最后将 is_loaded 标记为 True。feature_select 的功能是根据 select_feature 属性从 image_forward_outs 的隐藏状态中选择所需的特征。如果 select_feature 为 patch，则去掉第一个元素，也就是 cls_token；如果为 cls_patch，则保留全部；否则抛出异常。

```
    def forward(self, images):
        if type(images) is list:
            image_features = []
            for image in images:
                # print(image.shape)
                # import pdb; pdb.set_trace()
                image_forward_out = self.vision_tower(image.to(device=self.device, dtype=self.dtype).unsqueeze(0),
output_hidden_states=True)

                image_feature = self.feature_select(image_forward_out).to(image.dtype)
                # print(image_features.shape)

                image_features.append(image_feature)
        else:
            # print(images.shape)
            # import pdb; pdb.set_trace()
            image_forward_outs = self.vision_tower(images.to(device=self.device, dtype=self.dtype), output_hidden_states=True)
            image_features = self.feature_select(image_forward_outs).to(images.dtype)
            # print(image_features.shape)
```

最后是 forward 方法，它的功能是模型的前向传播函数。当输入 images 是列表时，对列表中的每个图像进行处理：将图像转换到相应设备和数据类型，通过 self.vision_tower 进行前向传播，使用 feature_select 提取特征并添加到 image_features 列表；当输入为单个图像时，直接进行相同操作，最终返回提取的图像特征。

2.2 文本部分

文本部分使用自定义的 geochat。

```
class GeoChatLlamaForCausalLM(LlamaForCausalLM, GeoChatMetaForCausalLM):
    config_class = GeoChatConfig

    def __init__(self, config):
        super(LlamaForCausalLM, self).__init__(config)
        self.model = GeoChatLlamaModel(config)

        self.lm_head = nn.Linear(config.hidden_size, config.vocab_size, bias=False)

        # Initialize weights and apply final processing
        self.post_init()

    def get_model(self):
        return self.model
```

代码定义了一个名为 GeoChatLlamaForCausalLM 的类，它继承自 LlamaForCausalLM 和 GeoChatMetaForCausalLM，主要用于实现一个因果语言模型 Causal Language Model，同时融合了地理信息，支持多模态输入。该类实现了模型的初始化、前向传播、生成输入准备等功能，还包括计算预测损失的逻辑。

首先是__init__方法，它调用 super(LlamaForCausalLM, self).__init__(config) 来调用父类 LlamaForCausalLM 的构造函数进行一些初始化操作。后面创建一个 GeoChatLlamaModel 实例并将其存储在 self.model 中，这个模型将是整个架构的核心部分，用于处理输入并生成中间表示。然后定义一个线性层 self.lm_head，将隐藏层大小映射到词汇表大小，该线性层将用于生成下一个词的概率分布，且没有偏置项。最后调用 self.post_init() 进行权重初始化和最终处

理。

get_model 方法就是简单地返回存储在 self.model 中的 GeoChatLlamaModel 实例，方便外部调用获取该模型。

```
    def forward(
        self,
        input_ids: torch.LongTensor = None,
        attention_mask: Optional[torch.Tensor] = None,
        past_key_values: Optional[List[torch.FloatTensor]] = None,
        inputs_embeds: Optional[torch.FloatTensor] = None,
        labels: Optional[torch.LongTensor] = None,
        use_cache: Optional[bool] = None,
        output_attentions: Optional[bool] = None,
        output_hidden_states: Optional[bool] = None,
        images: Optional[torch.FloatTensor] = None,
        return_dict: Optional[bool] = None,
    ) -> Union[Tuple, CausalLMOutputWithPast]:
        output_attentions = output_attentions if output_attentions is not None else self.config.output_attentions
        output_hidden_states = (
            output_hidden_states if output_hidden_states is not None else self.config.output_hidden_states
        )
        return_dict = return_dict if return_dict is not None else self.config.use_return_dict

        input_ids, attention_mask, past_key_values, inputs_embeds, labels = self.prepare_inputs_labels_for_multimodal(input_ids,
attention_mask, past_key_values, labels, images)

        # decoder outputs consists of (dec_features, layer_state, dec_hidden, dec_attn)
        outputs = self.model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            past_key_values=past_key_values,
            inputs_embeds=inputs_embeds,
            use_cache=use_cache,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=return_dict
        )

        hidden_states = outputs[0]
        logits = self.lm_head(hidden_states)

        loss = None
        if labels is not None:
            # Shift so that tokens < n predict n
            shift_logits = logits[..., :-1, :].contiguous()
            shift_labels = labels[..., 1:].contiguous()
            # Flatten the tokens
            loss_fct = CrossEntropyLoss()
            shift_logits = shift_logits.view(-1, self.config.vocab_size)
            shift_labels = shift_labels.view(-1)
            # Enable model/pipeline parallelism
            shift_labels = shift_labels.to(shift_logits.device)
            loss = loss_fct(shift_logits, shift_labels)

        if not return_dict:
            output = (logits,) + outputs[1:]
            return (loss,) + output if loss is not None else output

        return CausalLMOutputWithPast(
            loss=loss,
            logits=logits,
            past_key_values=outputs.past_key_values,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )
```

最重要的 forward 方法，它首先对一些参数，如 output_attentions、output_hidden_states 和 return_dict 进行处理，如果它们没有被显式提供，则使用配置文件中的默认值。然后调用 self.prepare_inputs_labels_for_multimodal 函数对输入进行预处理，处理可能包括对输入的转换、图像和文本信息的结合等，以便于后续处理。接着调用 self.model 的 forward 方法对处理后的输入进行前向传播，得到输出结果。将 self.model 的输出结果中的第一个元素作为 hidden_states，通过 self.lm_head 线性层将其转换为 logits。后面分析，如果提供了 labels，则计算预测的 loss，将 logits 和 labels 进行偏移操作，让 logits 预

测的是下一个词的概率，然后将 shift_logits 和 shift_labels 展开为 2D 张量，接着将 shift_labels 转移到 shift_logits 所在的设备，最后使用 CrossEntropyLoss 计算交叉熵损失。

根据 return_dict 参数决定返回结果的格式，如果 return_dict 为 False，将 loss 和其他输出拼接成元组返回；如果为 True，将结果包装在 CausalLMOutputWithPast 对象中返回。

```python
def prepare_inputs_for_generation(
    self, input_ids, past_key_values=None, attention_mask=None, inputs_embeds=None, **kwargs
):
    if past_key_values:
        input_ids = input_ids[:, -1:]

    # if `inputs_embeds` are passed, we only want to use them in the 1st generation step
    if inputs_embeds is not None and past_key_values is None:
        model_inputs = {"inputs_embeds": inputs_embeds}
    else:
        model_inputs = {"input_ids": input_ids}

    model_inputs.update(
        {
            "past_key_values": past_key_values,
            "use_cache": kwargs.get("use_cache"),
            "attention_mask": attention_mask,
            "images": kwargs.get("images", None),
        }
    )
    return model_inputs
```

最后是 prepare_inputs_for_generation 方法，它对生成过程的输入进行准备。如果 past_key_values 存在，将 input_ids 截取为最后一个元素，因为在生成任务中，通常是基于上一个生成的词来预测下一个词。根据 inputs_embeds 是否存在以及 past_key_values 是否为空，决定使用 input_ids 还是 inputs_embeds 作为输入。将 past_key_values、use_cache、attention_mask 和 images 等信息添加到 model_inputs 字典中，方便后续调用。

2.3 图文对齐头

图文对齐头采用的 LLaVA 方案。

```python
def build_vision_projector(config, delay_load=False, **kwargs):
    projector_type = getattr(config, 'mm_projector_type', 'linear')

    if projector_type == 'linear':
        return nn.Linear(config.mm_hidden_size, config.hidden_size)

    mlp_gelu_match = re.match(r'^mlp(\d+)x_gelu$', projector_type)
    if mlp_gelu_match:
        mlp_depth = int(mlp_gelu_match.group(1))
        modules = [nn.Linear(config.mm_hidden_size, config.hidden_size)]
        for _ in range(1, mlp_depth):
            modules.append(nn.GELU())
            modules.append(nn.Linear(config.hidden_size, config.hidden_size))
        return nn.Sequential(*modules)

    if projector_type == 'identity':
        return IdentityMap()

    raise ValueError(f'Unknown projector type: {projector_type}')
```

这个函数函数 build_vision_projector 的主要功能是根据配置信息创建一个图文对齐头，用于将输入从一个特征空间投影到另一个特征空间。它支持不同类型的投影器，如线性投影器、MLP 投影器，带有 GELU 激活函数和恒等映射投影器。它首先从配置对象 config 中获取 mm_projector_type 属性作为 projector_type，如果该属性不存在，则默认为 linear。如果 projector_type 是 linear，则创建一个 nn.Linear 线性层作为投影器，将输入维度从 config.mm_hidden_size 转换为 config.hidden_size。使用正则表达式 re.match(r'^mlp(\d+)x_gelu$', projector_type) 来检查 projector_type 是否符合 mlp<数字>x_gelu 的格式，其中 <数字> 表示 MLP 的深度，一般为 2。如果匹配成功，将 <数字> 部分提取出来并转换为整数 mlp_depth。然后创建一个模块列表 modules，首先添加一个 nn.Linear 层将输入维度从 config.mm_hidden_size 转换为 config.hidden_size。对于后续的 MLP 层（除了第一层），添加 nn.GELU 激活函数和 nn.Linear 层（维度保持为 config.hidden_size），并使用 nn.Sequential(*modules) 将这些模块组合在一起作为最终的投影器。

3 算法缺陷分析

3.1 数据集范围窄

GeoChat 在遥感图像领域的设计是基于特定的遥感数据集进行微调和训练的。该数据集的范围较窄（例如，仅仅包含某些特定类型的遥感图像），模型在推理其他类型遥感数据（如不同地理区域）时表现不佳。这意味着，GeoChat 可能无法适应新的、未知的遥感任务或数据。例如，GeoChat 可能在检测城市区域的遥感图像时表现良好，但对农业或森林等自然景观的遥感图像处理不够准确。

3.2 图文对齐效果不佳

GeoChat 的图文对齐能力依赖于将遥感图像和对应的文本信息（如描述、标签、问题等）在同一个空间中进行匹配。当图像和文本之间的对齐效果不佳时，模型可能会错误地理解文本与图像的关系，导致推理结果不准确。例如，在处理带有复杂地理信息的遥感图像时，模型可能无法正确识别文本描述中的特定区域，或者错误地将某些文本内容与图像中的区域匹配。

3.3 文本处理的 Token 过多

在 GeoChat 的架构中，文本处理是通过将图像信息转化为文本描述来完成的。在一些复杂任务中，模型可能需要处理大量的文本输入和输出，这会导致输入文本的 token 数量非常庞大。例如，在进行遥感图像的区域描述时，如果图像内容复杂，描述文本可能包含大量细节，这导致输入 token 过多，从而增加了模型计算的负担并可能影响推理效率。

4 改进思路

多样化数据集的引入与数据增强是提升 GeoChat 在遥感图像任务中的适应性和泛化能力的一个有效方案。通过引入更广泛的遥感数据集和进行数据增强，能够让模型在训练时学习到更广泛的特征，从而更好地处理各种不同类型的遥感图像。通过多样化数据集引入，模型能够学习到更多地物类型、地理特征和传感器数据；而通过数据增强，模型则能够从不同角度、不同季节、不同光照等条件下学习，使其更具泛化能力。
例如，可以在 GeoChat 中引入来自不同地理区域的农业遥感数据集时，数据增强，如对比度调整、旋转、缩放，可以帮助模型更好地适应不同气候条件、不同生长季节的农业区域图像。在训练过程中，通过不断扩展数据的多样性，GeoChat 模型将能够在面临不同类型的遥感任务时，表现出更强的适应性和准确性。

5 改进前后的对比实验分析

武汉大学遥感照片：

 （图1）

 （图 2）

效果对比：

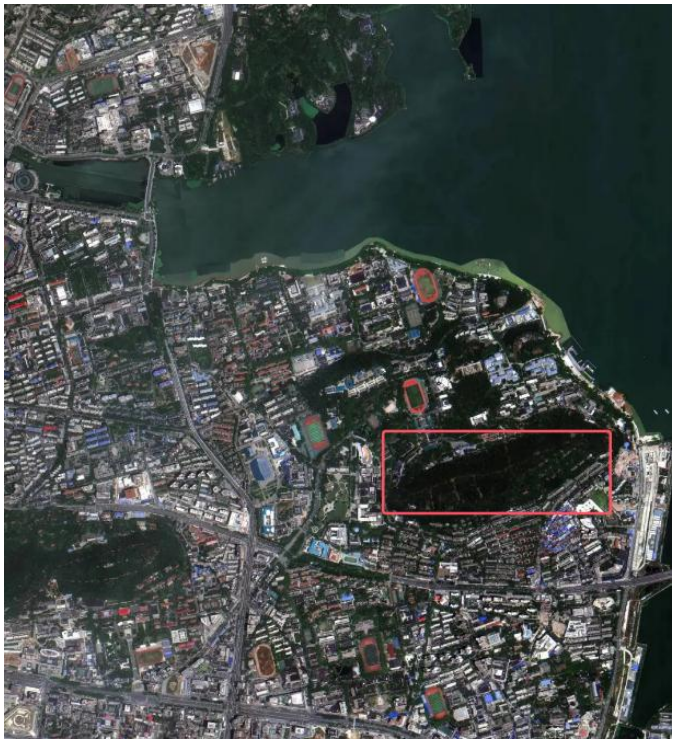
Question：[refer] where is the playground

Answer:

改进后效果：

Question: where is the biggest forest?
Answer:

改进后效果：



Question:Where is the forest?
Answer：

改进后效果：

Question:describe the building in the middle of the picture

Answer:

改进后：