MyTransformers项目代码使用文档

训练代码

训练入口

文件地址 **MyTransformers/train/u_train.py**

参数获取与准备分布式环境

```python
args = ds_parser(train_parser(base_parser())).parse_args()
# If args.test_code, the log file and tb writer will not be created.
if args.test_code:
    os.environ['NO_LOG_FILE'] = 'true'
args = registry.get_paths(args)
set_random_seed(args.seed)
device, args = init_dist(args)
```

模型获取

模型获取分为两类，如果是huggingface模型则通过load_huggingface_model获取，如果是本地模型（也就是模型代码在本地）则通过load_local_model获取。

```python
if args.huggingface:
    model, tokenizer, model_config = load_huggingface_model(args)
else:
    model, tokenizer, model_config = load_local_model(args)
```

现在模型训练一般使用lora训练，在获取模型之后还需要将模型中需要使用lora训练的层替换为lora层，并将可训练参数设置为可训练，不可训练参数设置为不可训练。

```python
if args.use_lora or args.use_lora_plus:
    if args.replace_modules is None:
        args.replace_modules = model_config.lora_layers
    switch_to_lora(model, args.replace_modules, rank=args.lora_rank)
    if args.lora_fa:
        lora_weight = ['weight_b']
    else:
        lora_weight = ['weight_a','weight_b']
    args.enable_list = lora_weight if args.enable_list is None else list(set(args.enable_list + lora_weight))
if args.enable_list is not None:
    enable_trainable_params(model, args.enable_list)
elif args.disable_list is not None:
    disable_untrainable_params(model, args.disable_list)
print_trainable_module_names(model)
```

数据集准备

和常用代码相同，准备数据集再使用dataloader加载（现在用的都是可迭代式数据集）。

```
data_collator = PipeLine_Datacollator(tokenizer) if args.num_pp_stages else DataCollator(tokenizer)
dataset_args = dict(mode=args.mode,
                    global_rank=args.global_rank,
                    meta_prompt=args.meta_prompt,
                    prefix=args.prefix,
                    postfix=args.postfix)

train_dataset = IterableDataset(args.train_dataset_path,
                                tokenizer,
                                max_len=args.max_len,
                                max_src_len=args.max_src_len,
                                read_nums=args.read_nums,
                                shuffle=True,
                                num_dp_ranks=parallel_states.get_data_parallel_world_size(),
                                dp_rank=parallel_states.get_data_parallel_rank(),
                                **dataset_args)
g = torch.Generator()
train_dataloader = DataLoader(train_dataset,
                              collate_fn=data_collator,
                              shuffle=False,
                              drop_last=True,
                              batch_size=args.batch_size_per_gpu,
                              generator=g)
```

模型训练

本仓库仅允许使用deepspeed进行训练，所以首先需要将模型包装为deepspeed引擎。

```python
optimizer, lr_scheduler = get_optimizer(ds_config, args, model=model)
engine, optimizer, _, _ = deepspeed.initialize(model=model,
                                               optimizer=optimizer,
                                               lr_scheduler=lr_scheduler,
                                               config=ds_config,
                                               model_parameters=[p for p in model.parameters() if p.requires_grad],
                                               mpu=None if args.num_pp_stages else parallel_states)
```

包装模型之后，需要准备forward_step/backward_step/eval_step，以及特定任务需要的task_print(训练默认只打印loss的信息，如果需要打印mcc等信息则需要准备)。

```python
if args.num_pp_stages:
    forward_step = forward_step_pipeline
    eval_step = eval_step_pipeline
    backward_step = None
    task_print = task_print_pipeline
else:
    forward_step = forward_step_deepspeed
    eval_step = eval_step_deepspeed
    backward_step = backward_step_deepspeed
    task_print = task_print_deepspeed

trainer = Trainer(args, writer)
trainer.register_task_print(task_print)
args.gradient_accumulation_steps = 1  # For correctly print info.

try:
    trainer.train(model=engine,
                  train_data_loader=train_dataloader_iter,
                  eval_data_loader=eval_dataloader_iter,
                  optimizer=None,
                  forward_step=forward_step,
                  backward_step=backward_step,
                  eval_step=eval_step,
                  log_loss=True)
except Exception as e:
    # When any error occurs during the training process, log the error.
    traceback_info = traceback.format_exc()
    print_rank_0(traceback_info, args.global_rank, logging.ERROR)
```

Trainer代码

huggingface模型训练一般使用hugginface的trainer库进行训练，同样的，本仓库也有自己的trainer代码，便于需要时进行修改。

文件地址 **MyTransformers/train/trainer.py**

trainer的初始化

初始化时需要传入args， args即是在训练代码中获取的参数。以及writer，该变量用于绘制tensorboard信息。

```python
def __init__(self, args, writer=None):
    self.args = args
    self.end = False
    self.writer = writer
    self.all_loss = 0.0
    self.global_step = 0
    self.all_metric = []
    self.eval_loss = None
    self.eval_metric = []
    self.best_eval_index = 0.0
    self.wait = 0
    self.save_floder = os.path.join(args.output_path, args.experiment_name)
    ensure_directory_exists(self.save_floder, self.args.global_rank)

```

训练过程

训练时需要传入的必要参数有

model：被训练的模型

forward_step: 模型前向传播的过程

backward_step: 模型反向传播的过程（deepspeed将反向传播和前向传播封装在了一起，所以这个参数可以不提供）

optimizer：优化器（使用deepspeed训练时不需要提供）

train_data_loader和eval_data_loader：数据集

eval_step: 模型进行评估时的过程

```python
def train(self,
          model:torch.nn.Module,
          train_data_loader:DataLoader,
          forward_step: Callable,
          optimizer: Callable = None,
          eval_data_loader:DataLoader = None,
          backward_step: Callable = None,
          eval_step: Callable = None,
          log_loss: bool = False):
```

具体的训练代码可以简单拆分为，前向传播->反向传播->在特定步数下跑验证集->打印训练信息->在特定步数下保存模型

```
    with Timer(iterations=total_print) as timer:
        for step in range(self.args.num_update_steps):
            timer.average_time(entry='start')
            # Execute the forward step of the model and calculate the loss and metric.
            loss, metric = forward_step(model, train_data_loader, self.args, step)
            # Accumulate loss and metric for gradient accumulation.
            self.all_loss += loss.item()
            self.all_metric.append(metric)
            # Execute the backward step if provided (optional) to update model parameters.
            if backward_step:
                backward_step(model, optimizer, loss)
            # Manage and print training information.
            # Evaluate the model at intervals specified by eval_interval.
            if (step + 1) % self.args.eval_interval == 0 and eval_step is not None:
                assert eval_data_loader is not None, 'evaluation dataset can not be None'
                self.eval_loss, eval_metric = eval_step(model, eval_data_loader, self.args, step)
                self.eval_metric.append(eval_metric)
            self.info_manager(step, timer, log_loss)
            self.save_model(model, step)
            if self.end:
                print_rank_0("Early stopping triggered.", self.args.global_rank)
                break
```

模型权重保存

由于流水线并行时模型是分层保存的，而数据并行时模型是完整保存的。所以使用流水线并行训练模型时，模型的保存过程需要特殊处理。deepspeed为流水线并行模型提供了保存方法，即model.save_checkpoint。至于流水线模型则直接使用torch_save方法即可。

```python
    def save_model(self, model, step):
        config_path =  os.path.join(self.save_floder, 'config.json')
        if not os.path.exists(config_path) and isinstance(self.args, Namespace):
            with open(config_path, 'w', encoding='utf-8') as f:
                print_rank_0(f'--->Saving training config at {step+1}th step in {config_path}.', self.args.global_rank)
                json.dump(self.args.__dict__, f)

        if self.args.num_pp_stages is not None:
            if not self.end and (step+1) % self.args.save_interval == 0:
                print_rank_0(f'--->Start saving model at {step+1}th step in {self.save_floder}.', self.args.global_rank)
                model.save_checkpoint(self.save_floder, tag=f'step_{step+1}')
                print_rank_0('--->Saved the model.', self.args.global_rank)
            elif self.end and (step+1) % self.args.save_interval != 0:
                print_rank_0(f'--->Start saving model at final step in {self.save_floder}.', self.args.global_rank)
                model.save_checkpoint(self.save_floder, tag='final')
                print_rank_0('--->Saved the model.', self.args.global_rank)
            return

        if self.args.global_rank <= 0:
            if not self.end and (step+1) % self.args.save_interval == 0:
                save_path = os.path.join(self.save_floder, f'step_{step+1}.ckpt')
                print_rank_0(f'--->Start saving model at {step+1}th step in {save_path}.', self.args.global_rank)
                self.torch_save(model, save_path)
                print_rank_0('--->Saved the model.', self.args.global_rank)
            elif self.end:
                save_path = os.path.join(self.save_floder, 'final.ckpt')
                print_rank_0(f'--->Start saving model at final step in {save_path}.', self.args.global_rank)
                self.torch_save(model, save_path)
                print_rank_0('--->Saved the model.', self.args.global_rank)
```

torch_save方法将torch.save再次包装了一层，能够很方便的只保存可训练参数。（通过参数中的save_trainable参数指定）

```python
    def torch_save(self, model:torch.nn.Module, save_path):
        if self.args.save_trainable:
            trainable_params = {}
            for name, param in model.module.named_parameters():
                if param.requires_grad:
                    trainable_params[name] = param.data
            torch.save(trainable_params, save_path)
        else:
            torch.save(model.module.state_dict(), save_path)
```

模型代码

本处使用llama的代码为例

代码地址 **MyTransformers/model/llama**

本仓库的模型需要提供四个文件：

模型配置

即config.py文件。该文件里需要提供模型的基本参数信息以dataclass类进行封装，形式如下：

```python
@dataclass
class ModelArgs:
    dim: int = 4096
    head_dim: int = 128
    hidden_size: int = 4096
    n_layers: int = 32
    num_hidden_layers: int = 32
    n_heads: int = 32
    num_attention_heads: int = 32
    n_kv_heads: Optional[int] = None
    vocab_size: int = -1  # defined later by tokenizer
    multiple_of: int = 256  # make SwiGLU hidden layer size multiple of large power of 2
    ffn_dim_multiplier: Optional[float] = None
    norm_eps: float = 1e-5
    rope_theta: float = 100000.0
    max_batch_size: int = 32
    max_seq_len: int = 2048
    tokenizer: Optional[str] = ''
    lora_layers: List[str] = field(default_factory=lambda: ['wk', 'wv', 'wq', 'wo', 'w1', 'w2', 'w3'])
    dtype: str = 'float16'
    def get_dtype(self) -> Optional[torch.dtype]:
        """Gets the torch dtype from the config dtype string."""
        return STR_DTYPE_TO_TORCH_DTYPE.get(self.dtype, None)
```

还需要提供不同大小的模型需要的不同参数配置，并使用注册器进行注册，形式如下：

```python
@registry.register_model_config("llama3_8b")
def get_config_for_llama3_8b() -> ModelArgs:
    return ModelArgs(
        n_kv_heads=8,
        multiple_of=1024,
        ffn_dim_multiplier=1.3,
        vocab_size=128256,
        rope_theta=500000.0
    )
```

模型

即model.py文件。该文件里需要提供模型的定义代码和inference代码，一般来说从开源代码中copy并稍微进行改动即可，最后要使用注册器进行注册，形式如下：

```python
@registry.register_model("llama3")
class Llama3(LlamaGenerate):
    def __init__(self, model_args: ModelArgs):
        try:
            self.tokenizer = Llama3Tokenizer(model_args.tokenizer)
            model_args.vocab_size = self.tokenizer.n_words
        except:
            pass
        super().__init__(model_args=model_args)
```

模型数据并行类

即train_model.py文件，由于开源模型代码一般仅仅提供定义代码和inference代码，所以要把模型用于训练的话还需要另外写一个训练类，将模型训练时的数据流动进行包装。

需要注意的是，不同模型的数据并行类都必须只接受model和args两个参数，如果需要修改，则要把每个模型的代码都进行修改。并在u_train代码中做对应的修改。

该类需要继承BaseModel类，并使用注册器进行注册，形式如下：

```python
@registry.register_train_model('llama3')
@registry.register_train_model('llama')
class LLaMaTrainModel(BaseModel):
    """
    Trainer class for llama, responsible for handling input and output during training.
    """
    def __init__(self, model:LlamaGenerate, args):
        """
        Initializes basic attributes for the trainer class and precomputes fixed values.

        param model: llama model with pretrained weight.
        param args: Arguments from argument parser.
        """
        super().__init__(args)
        self.layers = model.model.layers
        self.tok_embeddings = model.model.tok_embeddings
        self.output = model.model.output
        self.norm = model.model.norm
        self.attention_mask = LLaMaTrainModel.get_masks(args.max_len)
        self.freqs_cis = precompute_freqs_cis(args.head_dim,
                                              args.max_len,
                                              theta=args.rope_theta,
                                              train_pi=args.train_pi,
                                              train_pipeline=False)
```

前向过程直接继承BaseModel类即可：

```python
    def forward(self, **kwargs):
        return super().forward(**kwargs)
```

但是该类需要额外提供三个方法：

embedding方法，用于语言模型的嵌入，该方法会在forward方法中被调用

```python
    def embedding(self, input_ids):
        hidden_states = self.tok_embeddings(input_ids)
        attention_mask = self.attention_mask.to(hidden_states.device).to(hidden_states.dtype)
        return hidden_states, attention_mask
```

model_forward方法，用于模型的前向传播过程，该方法同样会在forward方法中被调用

```python
    def model_forward(self, logits, freqs_cis, attention_mask):
        # Using activation checkpoint to reduce memory consumption or not.
        for i in range(self.args.num_layers):
            if self.args.activation_checkpoint:
                logits = checkpoint(self.layers[i],
                                    logits,
                                    0,
                                    freqs_cis,
                                    attention_mask,
                                    self.args.atten_type,
                                    use_reentrant=False)
            else:
                logits = self.layers[i](x=logits,
                                        start_pos=0,
                                        freqs_cis=freqs_cis,
                                        mask=attention_mask,
                                        atten_type=self.args.atten_type)
        logits = self.norm(logits)
        logits = self.output(logits)
        return logits
```

get_masks静态方法，用于获取attention mask，该方法会在多个地方被使用（包括类的外部），所以需要写为静态方法

```
    @staticmethod
    def get_masks(seqlen, device='cpu', dtype=torch.float, start_pos=0):
        if seqlen > 1:
            mask = torch.full((seqlen, seqlen), float("-inf"), device=device)
            mask = torch.triu(mask, diagonal=1)
            mask = torch.hstack([torch.zeros((seqlen, start_pos), device=device),mask]).to(dtype)
            return mask
```

模型流水线并行类

即pipeline_model.py，该类专门用于流水线并行训练

需要编写模型中每一个逻辑层的代码，并封装在一起，最后使用注册器注册，形式如下：

```
@registry.register_pipeline_model("llama3")
@registry.register_pipeline_model("llama")
def get_pipeline_model(model, args):
    layers = [LayerSpec(EmbeddingPipelineLayer, model=model, args=args),
            *[LayerSpec(DecoderPipelineLayer, model=model, args=args, layer_idx=idx) for idx in
            range(args.num_layers)],
            LayerSpec(FNormPipelineLayer, model=model),
            LayerSpec(LossPipelineLayer, pad_id=args.pad_id)]
    return PipelineModule(layers=layers, num_stages=args.num_pp_stages, partition_method='uniform')
```

逻辑层的代码形式如下，模型的每一个逻辑部分都需要写成这样的逻辑层，其输入是一个模型，并在这个模型中提取需要的部分，然后处理该部分的输入输出。其forward函数需要遵守deepspeed的协议，仅接受一个inputs输入元组，进入具体逻辑后再进行拆分:

```python
class DecoderPipelineLayer(torch.nn.Module):
    def __init__(self, model: LlamaGenerate, layer_idx, args):
        super().__init__()
        self.layer = model.model.layers[layer_idx]
        self.args = args

    def forward(self, inputs):
        hidden_states, freqs_cis, attention_mask, labels = inputs
        # [batch_size, input_len, hidden_dim]
        if self.args.activation_checkpoint:
            hidden_states = checkpoint(self.layer,
                                       hidden_states,
                                       0,
                                       freqs_cis,
                                       attention_mask,
                                       self.args.atten_type,
                                       use_reentrant=False)
        else:
            hidden_states = self.layer(hidden_states,
                                       0,
                                       freqs_cis,
                                       attention_mask,
                                       self.args.atten_type)
        return hidden_states, freqs_cis, attention_mask, labels
```

通用代码

由于每个模型都有共通的部分，所以这些内容可以单独抽象出来

模型数据并行基类

即base_model.py文件，该文件提供BaseModel类，该类提供了一个模型训练的基本流程，不同的模型只需要编写模型特定的子流程即可。

```python
    def forward(self, **kwargs):
        input_ids, labels = kwargs["input_ids"], kwargs["labels"]
        input_ids, labels, freqs_cis = self.cut_sequence(input_ids, labels)
        hidden_states, attention_mask = self.embedding(input_ids)
        logits = self.model_forward(hidden_states, freqs_cis, attention_mask)
        loss = self.compute_loss(logits, labels)
        return loss, self.compute_metric(logits, labels, kwargs["cal_metric_pos_tensor"])
```

具体的模型需要完成编写子流程，否则会导致NotImplementedError。

```
    def embedding(self, input_ids):
        raise NotImplementedError()

    def model_forward(self, hidden_states, freqs_cis, attention_mask):
        raise NotImplementedError()
```

该文件还提供了一些每个模型都要使用的子流程，如loss的计算。具体的模型如果需要修改loss的计算，那么直接覆盖这个方法即可。

```
    def compute_loss(self, logits, labels):
        shift_logits = logits[..., :-1, :].contiguous()
        shift_labels = labels[..., 1:].contiguous()
        loss = self.loss_fct(shift_logits.reshape(-1, shift_logits.size(-1)), shift_labels.reshape(-1))
        return loss
```

注意力机制代码

即attention.py，在模型的训练过程中，可能会需要使用到flash_attention等代码，而不同模型的注意力机制基本是相同的，所以把这部分抽象出来，方便对注意力机制进行不同的处理。

```
def attention_func(
   q: torch.Tensor,
   k: torch.Tensor,
   v: torch.Tensor,
   atten_mask: torch.Tensor,
   dropout_p: float,
   scaling: float,
   is_causal: bool,
   atten_type: str = ''
):
    """
    Attention function that supports different attention types.

    Args:
        q (torch.Tensor): Query tensor of shape [batch_size, n_local_heads, input_len, head_dim].
        k (torch.Tensor): Key tensor of shape [batch_size, n_local_heads, input_len, head_dim].
        v (torch.Tensor): Value tensor of shape [batch_size, n_local_heads, input_len, head_dim].
        atten_mask (torch.Tensor): Attention mask tensor of shape [batch_size, 1, input_len, input_len].
        dropout_p (float): Dropout probability.
        scaling (float): Scaling factor for the attention scores.
        is_causal (bool): Whether the attention is causal (only attend to past tokens).
        atten_type (str, optional): Type of attention to use. Can be 'flash_atten', 'ulysses_flash_atten', 'ulysses_atten', or leave empty for the default naive_attention_func.

    Returns:
        torch.Tensor: Output tensor of shape [batch_size, n_local_heads, input_len, head_dim].
    """
    if atten_type == 'flash_atten':
        require_version("torch>=2.0.0")
        with torch.backends.cuda.sdp_kernel(enable_flash=True):
            output = F.scaled_dot_product_attention(q, k, v, attn_mask=atten_mask, dropout_p=dropout_p, is_causal=is_causal)
    elif atten_type == 'ulysses_flash_atten':
        require_version("torch>=2.0.0")
        with torch.backends.cuda.sdp_kernel(enable_flash=True):
            flash_atten = F.scaled_dot_product_attention
            dist_atten = DistributedAttention(flash_atten, parallel_states.get_sequence_parallel_group(), scatter_idx=1, gather_idx=2)
            output = dist_atten(q, k, v, attn_mask=atten_mask, dropout_p=dropout_p, is_causal=is_causal)
    elif atten_type == 'ulysses_atten':
        dist_atten = DistributedAttention(naive_attention_func, parallel_states.get_sequence_parallel_group(), scatter_idx=1, gather_idx=2)
        output = dist_atten(q, k, v, atten_mask=atten_mask, dropout_p=dropout_p, scaling=scaling, is_causal=is_causal)
    else:
        output = naive_attention_func(q, k, v, atten_mask=atten_mask, dropout_p=dropout_p, scaling=scaling, is_causal=is_causal)
    return output
```

Common库

Lora

因为peft库用起来有点麻烦，而且也不得心应手，为了方便修改和使用，我自己写了一个lora的代码。

文件地址 MyTransformers/common/lora.py。

该库中提供了LinearWithLoRA类，可以将任何Linear层封装到这个类中，就能实现使用Lora训练。该类同时兼容了dora和plora的代码，如果有需要，可以将其他的lora变体的逻辑补充到这个类中。

```python
class LinearWithLoRA(nn.Linear):
    def __init__(
        self,
        in_features: int,
        out_features: int,
        lora_rank: int = 4,
        lora_scaler: float = 32.0,
        use_dora: bool = False,
        quant: bool = False,
        plora_steps: Union[int, None] = None
    ):
```

具体模型训练时，直接调用该文件中提供的switch_to_lora函数即可，需要提供以下参数：

model：需要使用lora训练的模型

replace_names: 需要使用lora替换的层，如['wq','wv','wo']表示将模型中所有名字包含这三个字符串的线性层替换为lora层。

rank：lora的秩

lora_scaler: lora的缩放因子

transposition：torch的线性层代码如linear_1(x)的实际逻辑为x*linear_1.weight.T，但是有些模型的代码的线性层逻辑可能写为x*linear_1.weight，在这种时候，参数矩阵就需要转置，才能放入lora层。

该代码首先找到需要被替换的层，确保该层具有in_features和out_features两个参数，确保lora层的维度和原来的层的唯独相同，再把相关的参数带入LinearWithLoRA类中构成一个lora层的实例。然后复制原来的参数到lora层中。

最后需要找到这个层的父层，再进行替换

```python
def switch_to_lora(model:nn.Module,
                   replace_names:Optional[Union[str, List[str]]] = None,
                   rank:int = 4,
                   lora_scaler:int = 32,
                   transposition:bool = False,
                   use_dora:bool = False,
                   plora_steps: Optional[int] = None):
```

```python
    """
    Switch function for lora, responsible for replacing Linear layer with LinearWithLoRA layer

    param model: Any pytorch model.
    param replace_names: List of module names to be replaced by LoRA.
    param rank: Rank for LoRA.
    param lora_scaler: Scaler for LoRA.
    param transposition: nn.Linear(x, w) compute xw^T, so the weight should in shape [out_feature, in_feature]. Otherwise, transposition should be set to True
    param use_dora: Weather to use dora
    param plora_steps: The steps to merge and reset lora weight.
    """
    assert replace_names is not None, 'Replace names can not be None'
    for name, module in model.named_modules():
        replace_tag = False
        for replace_name in replace_names:
            if replace_name in name:
                # Create LoRA layer instance.
                replace_tag = True
                if isinstance(module, LinearWithLoRA):
                    module.merge_and_reset(new_rank=rank)
                elif isinstance(module, nn.Module):
                    assert all(hasattr(module, attr) for attr in ["in_features", "out_features", "weight"]), \
                    "Module is missing one or more of the required attributes: 'in_features', 'out_features', 'weight'"
                    quant = getattr(module, "quant", False)
                    lora_layer = LinearWithLoRA(lora_rank=rank,
                                                lora_scaler=lora_scaler,
                                                in_features=module.in_features,
                                                out_features=module.out_features,
                                                use_dora=use_dora,
                                                quant=quant,
                                                plora_steps=plora_steps)
                    # Copy the original weight to the LoRA layer.
                    if transposition:
                        lora_layer.weight = nn.Parameter(module.weight.data.T)
                    else:
                        lora_layer.weight.data = module.weight.data
                    if quant:
                        lora_layer.weight_scaler = module.weight_scaler
                    # Replace the original layer with the LoRA layer.
                    parent = get_parent_model(model, module)
```

```python
          setattr(parent, list(parent._modules.items())[list(parent._modules.values()).index(module)][0], lora_layer)
    if not replace_tag and isinstance(module, LinearWithLoRA):
      # Merge weight to avoid unnecessary computing.
      module.merge_and_del()
```

找父层的过程基本就是一个深度遍历的过程（比二叉树的遍历稍微简单一些），主要是需要找到该层的直接父层。

```python
def get_parent_model(parent_model, module):
    """
    Find the parent module for the input module recursively.

    param parent_model: Root model for the search.
    param module: Submodule to find the parent module for.

    Returns:
    Parent module if found, None otherwise.
    """
    for _, sub_module in parent_model._modules.items():
        # Direct sub modules of parent model.
        if sub_module is module:
            return parent_model
        parent = get_parent_model(sub_module, module)
        if parent:
            return parent
    return None
```

Parser

parser代码用于准备模型训练过程中所需要的参数，如学习率等。parser中有三个子parser：

- base_parser
- train_parser
- ds_parser

所以使用时需要这样使用：ds_parser(train_parser(base_parser())).get_args()

注册器

注册器是本库中非常重要的特性，考虑到本库的目标是兼容多个不同的模型，所以为了避免适应不同的模型而在模型训练过程中产生过多冗余的代码，我采用了注册器机制。

如以下代码中，获取tokenizer，模型参数，模型定义，模型训练类，都需要通过注册器来完成。使用一个注册器来适应多个不同的模型，而不需要为每个模型准备一段代码。(但是每个模型的接口都需要

一致才能使用)

```python
def load_local_model(args):
    # Train with local defined model.
    tokenizer = registry.get_tokenizer_class(args.tokenizer_name)(args.tokenizer_path)
    print_rank_0(f'--->Using tokenizer: {args.tokenizer_name} with path: {args.tokenizer_path}', args.global_rank)
    config_type = '_'.join([args.model_name, args.variant])
    model_config = registry.get_model_config_class(config_type)()
    print_rank_0(f'--->Using model config: {config_type}', args.global_rank)
    model_config.vocab_size = tokenizer.n_words
    model = registry.get_model_class(args.model_name)(model_config)
    print_rank_0(f'--->Using model: {args.model_name}, and loading its trainning variant', args.global_rank)

    # Load checkpoint if checkpoint path is provieded.
    if args.ckpt_path is not None and args.from_pretrained:
        load_ckpt(model=model.model, ckpt_path=args.ckpt_path, partial_ckpt_path=args.partial_ckpt_path, rank=args.global_rank)
        print_rank_0(f'--->Using pretrained checkpoint at {args.ckpt_path}', args.global_rank)
    else:
        print_rank_0('--->Not using pretrained checkpoint to start traning.', args.global_rank)

    # Load config from model config to argument parser namespace.
    args.head_dim = model_config.head_dim
    args.head_num = model_config.num_attention_heads
    args.hidden_size = model_config.hidden_size
    args.num_layers = model_config.num_hidden_layers
    args.rope_theta = model_config.rope_theta if args.rope_theta is None else args.rope_theta
    args.pad_id = tokenizer.pad_id

    # Load model to training dtype.
    if args.fp16:
        model.half()
    elif args.bf16:
        model.bfloat16()

    # Convert model to trainable model for given training type.
    if args.num_pp_stages:
        pipe_model_cls = registry.get_pipeline_model_class(args.model_name)
        model = pipe_model_cls(model, args)
    else:
        train_model_cls = registry.get_train_model_class(args.model_name)
        model = train_model_cls(model, args)
    return model, tokenizer, model_config

```

注册器本身是一个包含了一个映射字典，和多个注册与取出的静态装饰器方法的类：

```
class Regitry:
    mapping = {
        "model_mapping":{},
        "pipeline_model_mapping":{},
        "train_model_mapping":{},
        "model_config_mapping":{},
        "dataset_mapping":{},
        "info_manager_mapping":{},
        "tokenizer_mapping":{},
        "paths_mapping":{}
    }

```

注册装饰器如下：

```
    @classmethod
    def register_model(cls, name):
        def wrap(model_cls):
            if model_cls in cls.mapping['model_mapping']:
                raise KeyError(
                    "Name '{}' already registered for {}.".format(
                        name, cls.mapping["model_mapping"][name]
                    )
                )
            cls.mapping['model_mapping'][name] = model_cls
            return model_cls
        return wrap

```

取定义的静态方法如下：

```
    @classmethod
    def get_model_class(cls, name):
        result = cls.mapping["model_mapping"].get(name, None)
```

```python
        if result is None:
            raise ValueError(f"Can not find name: {name} in model mapping, \
supported models are listed below:{cls.list_models()}")
        else:
            return result
```

Utils

utils库中包含了许多有用的函数和类

Timer: 这是一个用于计时的环境，通常用在循环上面，可以用于计算单次循环的用时，和整个循环的用时。

```python
class Timer(object):
    def __init__(self, start=None, n_round=2, iterations: Optional[int] = None):
```

logging配置：本库中的logging具有两个handler，一个是文件handler一个是控制台handler。文件handler负责将日志信息输出到文件中，而控制台handler负责将日志信息打印到控制台

handler的行为通过环境变量来控制，PRINTLEVEL负责控制控制台handler的级别，如设置为DEBUG，则级别低于DEBUG的日志不会被输出到控制台中

LOGLEVEL负责控制文件handler的级别，设置为INFO，则级别低于INFO的信息不会被写入文件

NO_LOG_FILE用于负责文件的创建，当测试代码时，可以通过这个变量避免产生文件

日志文件会根据当前时间进行创建，文件名即为时间。

```python
def configure_logging(log_path, rank: Optional[int]=0):
    """
    Configure logging functionality.

    Args:
    log_path (str): Path where the log files will be stored.
    rank (optional[int]): Level used for creating directories, default is 0.

    Returns:
    loggegr (logging.Logger): Configured logger object.
    """
    # Get environment variables.
    sh_level = os.environ.get("PRINTLEVEL", logging.DEBUG)
    fh_level = os.environ.get("LOGLEVEL", logging.INFO)
    fh_disable = os.environ.get("NO_LOG_FILE", "false") == 'true' # Convert string variable to boolean

    logger = logging.getLogger("MyTransformers")
    logger.setLevel(logging.DEBUG)
    formatter = logging.Formatter('[%(asctime)s] [%(levelname)s] %(message)s') # Define the log format

    # Create a console log handler
    sh = logging.StreamHandler()
    sh.setFormatter(formatter)
    sh.setLevel(sh_level)
    logger.addHandler(sh)

    timezone = pytz.timezone('Asia/Shanghai')
    # Get current date and time strings, formatted as 'yy-mm-dd' and 'HH-MM.log'
    date_string, hour_string = datetime.now().strftime('%y-%m-%d'), datetime.now().strftime('%H-%M') + '.log'
    log_path = os.path.join(log_path, date_string)

    ensure_directory_exists(log_path, rank)
    if not fh_disable:
        fh = logging.FileHandler(os.path.join(log_path, hour_string))
        fh.setFormatter(formatter)
        fh.setLevel(fh_level)
        logger.addHandler(fh)

    return logger
```

打印函数：本库中的print_rank_0会自动使用日志的形式进行打印，当log的级别合适时，print的内容会被输出到控制台，并写入文件中。

```
def print_rank_0(msg, rank=0, level=logging.INFO, flush=True):
    if "logger" not in globals() and rank<=0:
        global logger
        # Create logger when pring_rank_0 being used.
        logger = configure_logging(os.environ.get("LOG_FLODER", "log"), rank)
    if isinstance(level, str):
        level = getattr(logging, level.upper(), logging.INFO)
    if rank <= 0:
        logger.log(msg=msg, level=level)
        if flush:
            logger.handlers[0].flush()
```

加载模型的函数，load_ckpt可以结合预训练权重和后训练权重进行加载。后训练权重会被优先加载到模型中（覆盖掉预训练权重）。这个函数可以配合保存可训练参数的函数使用，这样训练时就能够只保存可训练函数，并正常加载模型。

并且该函数能够支持加载具有unexpected keys和missing keys的权重而不报错，但是会将相关的信息打印出来。

```python
def get_merged_state_dict(ckpt_path:str=None,
                          partial_ckpt_path:str=None):
    if ckpt_path:
        ckpt = torch.load(ckpt_path, map_location='cpu')
    else:
        ckpt = {}

    if partial_ckpt_path:
        partial_ckpt = torch.load(partial_ckpt_path, map_location='cpu')

        for name, param in partial_ckpt.items():
            ckpt[name] = param
    return ckpt

def load_ckpt(model:Module,
              ckpt_path:str=None,
              partial_ckpt_path:str=None,
              rank:int=0):
    """
    load model checkpoint safely.

    param model: Any pytorch model.
    param ckpt_path: Path of checkpoint of all params.
    param partial_ckpt_path: Path of patial model checkpoint. Must be provieded if load trainable params and pretrained params.
    """

    rank_0 = rank == 0
    model_state_dict = model.state_dict()
    ckpt = get_merged_state_dict(ckpt_path, partial_ckpt_path)

    try:
        model.load_state_dict(ckpt, strict=False)
    except Exception:
        if rank_0:
            print_exc()

    missing_keys = [key for key in model_state_dict.keys() if key not in ckpt.keys()]
    unexpected_keys = [key for key in ckpt.keys() if key not in model_state_dict.keys()]

    print_rank_0(f'--->Missing keys:{missing_keys}. Unexpected keys:{unexpected_keys}.', rank=rank)
```

初始化分布式环境

```python
def init_dist(args):
    if args.device == 'cuda':
        if args.local_rank == -1:
            device = torch.device("cuda")
        else:
            torch.cuda.set_device(args.local_rank)
            device = torch.device("cuda", args.local_rank)
            deepspeed.init_distributed(dist_backend="nccl")
            args.world_size = dist.get_world_size()
            args.global_rank = dist.get_rank()
    else:
        device = 'cpu'
    if args.num_sp_stages:
        assert args.atten_type == 'ulysses_atten', 'when using sequence parallism, the attention type must be `ulysses_atten`'
        parallel_states.initialize_model_parallel(sequence_model_parallel_size=args.num_sp_stages)
    elif args.num_pp_stages:
        parallel_states.initialize_model_parallel(pipeline_model_parallel_size=args.num_pp_stages)
    else:
        parallel_states.initialize_model_parallel()
    return device, args

```

随机种子

```python
def set_random_seed(seed):
    if seed is not None:
        random.seed(seed)
        np.random.seed(seed)
        torch.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
```

优化器

由于在一些情况下需要对优化器进行修改，使用deepspeed提供的优化器功能不太完整，需要自己写一些补充功能。

所以本库还提供了一个优化器的代码，地址 **MyTransformers/common/optimizer.py**

主要是一个获取优化器的函数，能够使用args获取不同的优化器。

```python
def get_optimizer(ds_config, args, model):
    # TODO: Add support to PLoRA and so no ...
    if args.diy_optimizer:

        # Acquire optimizer type.
        if args.optim_type is not None:
            optim_type = args.optim_type.lower()
        elif 'optimizer' in ds_config:
            optim_type = ds_config['optimizer'].get('type', 'adamw').lower()

        # Acquire optimizer.
        if args.use_galore:
            message = 'galore cannot be used with the current DeepSpeed version, and running it will result in an error.'
            warnings.warn(message, UserWarning)
            isSuccess, optimizer =  get_galore_optimizer(optim_type, args, model)
        else:
            isSuccess, optimizer =  get_regular_optimizer(optim_type, args, model)

        # If successful acquire optimizer, overwrite the default setting in the ds config.
        if isSuccess:
            del ds_config['optimizer']
            print_rank_0(F'--->deepspeed optimizer setting have been overwritten', args.global_rank)
        else:
            print_rank_0(f'--->try to use diy optimizer failed, use the ds setting', args.global_rank)

        lr_scheduler = get_learning_rate_scheduler(optimizer, 0, args)
        return optimizer, lr_scheduler

    return None, None
```

学习率规划器

这里的学习率规划器代码主要是从SwissArmyTransformer库中copy的，SAT库又是从英伟达那copy的

现在一般是使用cosine的方式对学习率进行规划，如果有新的学习率规划的方式也可以在这个代码上进行补充修改。

```python
class AnnealingLR(_LRScheduler):
    """Anneals the learning rate from start to zero along a cosine curve."""

    DECAY_STYLES = ['linear', 'cosine', 'exponential', 'constant', 'None']

    def __init__(self, optimizer, start_lr, warmup_iter, num_iters, decay_style=None, last_iter=-1, decay_ratio=0.5, auto_warmup_steps=50, auto_warmup_rate=0.05):
        """
        params start_lr: base learning rate for the scheduler
        params num_iters: total steps
        params warmup_iter: how much iters to use warmup
        params last_iters: how much iters already trained
        params auto_warmup_steps: the fix warmup step
        params auto_warmup_rate: the lr radio for fix warmup step
        params decay_style: learning rate decay style after warmup
        """
        assert warmup_iter <= num_iters
        self.optimizer = optimizer
        self.lr_scale = deepcopy([x['lr'] if 'lr' in x else 1. for x in optimizer.param_groups])
        self.start_lr = start_lr
        self.warmup_iter = warmup_iter
        self.init_step = last_iter
        self.num_iters = last_iter + 1
        self.end_iter = num_iters
        self.decay_style = decay_style.lower() if isinstance(decay_style, str) else None
        self.decay_ratio = 1 / decay_ratio
        self.auto_warmup_steps = auto_warmup_steps
        self.auto_warmup_rate = auto_warmup_rate
        self.step(self.num_iters)
        if not torch.distributed.is_initialized() or torch.distributed.get_rank() == 0:
            print_rank_0(f'--->learning rate decaying style {self.decay_style}, ratio {self.decay_ratio}')

    def get_lr(self):
        # auto_warmup_steps并不取决于warmup的设置，而是固定的进行warmup
        if self.num_iters <= self.init_step + self.auto_warmup_steps:
            auto_lr = float(self.start_lr) * self.auto_warmup_rate
            scheduled_lr = float(self.start_lr) * self.num_iters / self.warmup_iter
            return min(auto_lr, scheduled_lr)

        # 根据warmup设置进行warmup
        if self.warmup_iter > 0 and self.num_iters <= self.warmup_iter:
            return float(self.start_lr) * self.num_iters / self.warmup_iter
        else:
            # 进行learning rate decay
            if self.decay_style == self.DECAY_STYLES[0]:
                return self.start_lr*((self.end_iter-(self.num_iters-self.warmup_iter))/self.end_iter)
            elif self.decay_style == self.DECAY_STYLES[1]:
                decay_step_ratio = min(1.0, (self.num_iters - self.warmup_iter) / self.end_iter)
                return self.start_lr / self.decay_ratio * (
                        (math.cos(math.pi * decay_step_ratio) + 1) * (self.decay_ratio - 1) / 2 + 1)
            elif self.decay_style == self.DECAY_STYLES[2]:
                return self.start_lr
            else:
                return self.start_lr

    def step(self, step_num=None):
        if step_num is None:
            step_num = self.num_iters + 1
        self.num_iters = step_num
        new_lr = self.get_lr()
        for group, scale in zip(self.optimizer.param_groups, self.lr_scale):
            group['lr'] = new_lr * scale

    def state_dict(self):
        sd = {
                'start_lr': self.start_lr,
                'warmup_iter': self.warmup_iter,
                'num_iters': self.num_iters,
                'decay_style': self.decay_style,
                'end_iter': self.end_iter,
                'decay_ratio': self.decay_ratio
        }
        return sd
```