MindSpeed-Core-MS API Documentation

Release r0.1.0

MindSpeed-Core-MS

Dec 30, 2024

CONTENTS

CHAPTER

ONE

MINDSPEED-CORE-MS API

1.1 MindSpeed-Core-MS API

`mindspeed_ms.core.config.AllConfig`	A Config that contains all other configs, which will be used in init_configs methods as the default config.
`mindspeed_ms.core.config.DatasetConfig`	Dataset config class.
`mindspeed_ms.core.config.init_configs_from_yaml`	Initialize config class from configuration yaml file.
`mindspeed_ms.core.config.ModelParallelConfig`	Parallel config class.
`mindspeed_ms.core.config.MoEConfig`	MoE config class.
`mindspeed_ms.core.config.TrainingConfig`	Training config.
`mindspeed_ms.core.config.TransformerConfig`	Transformer config class.
`mindspeed_ms.core.parallel_state.get _context_parallel_rank`	Return rank for the context parallel group.
`mindspeed_ms.core.parallel_state.get _context_parallel_world_size`	Return world size for the context parallel group.
`mindspeed_ms.core.parallel_state.initialize _model_parallel`	Initialize model data parallel groups.
`mindspeed_ms.core.tensor_parallel.ColumnParallelLinear`	The dense layer with weight sliced on second dimension by tensor parallel size.
`mindspeed_ms.core.tensor_parallel.RowParallelLinear`	The dense layer with weight sliced on first dimension by tensor parallel size.
`mindspeed_ms.core.tensor_parallel.VocabParallelCrossEntropy`	Calculate the paralleled cross entropy loss.
`mindspeed_ms.core.tensor_parallel.VocabParallelEmbedding`	Embedding parallelized in the vocabulary dimension.
`mindspeed_ms.legacy.model.eos_mask.EosMask`	Generate attention mask corresponding to a specific token.
`mindspeed_ms.legacy.model.gpt_model.GPTModel`	The Generative Pre-trained Transformer (GPT) is a decoder-only Transformer model.
`mindspeed_ms.legacy.model.language_model.Embedding`	An embedding layer contain word embedding, position embedding and tokentypes embedding.
`mindspeed_ms.legacy.model.language_model.get_language_model`	Use this function to get language model.
`mindspeed_ms.legacy.model.module.Module`	Specific extensions of cell with support for pipelining.
`mindspeed_ms.legacy.model.moe.experts.SequentialMLP`	Define SequentialMLP module.

continues on next page

Table 1 – continued from previous page

mindspeed_ms.legacy.model.moe.moe_layer.MoELayer	Expert layer.
mindspeed_ms.legacy.model.moe.router.TopKRouter	TopK router.
mindspeed_ms.legacy.model.moe.token_dispatcher.MoEAlltoAllTokenDispatcher	In the MoE architecture, the MoEAlltoAllTokenDispatcher scheduler is responsible for assigning tokens to various experts for processing, and reassembling the processed results back to the original token order.
mindspeed_ms.legacy.model.ParallelAttention	This class represents a parallel attention mechanism.
mindspeed_ms.legacy.model.ParallelLMLogits	Head to get the logits of each token in the vocab.
mindspeed_ms.legacy.model.ParallelMLP	Implementation of parallel feedforward block.
mindspeed_ms.legacy.model.ParallelTransformer	A transformer block.
mindspeed_ms.legacy.model.ParallelTransformerLayer	A single transformer layer.
mindspeed_ms.legacy.model.RotaryEmbedding	Rotary positional embedding for language model.
mindspeed_ms.legacy.model.transformer.CoreAttention	Core attention mechanism for calculating attention weights and context representations between queries, key and value layers.
mindspeed_ms.legacy.model.TransformerLanguageModel	Transformer language model.
mindspeed_ms.training.loss_func.LossWithMask	Calculate the loss with mask and mean reduction.
mindspeed_ms.training.training.pretrain	Pre-training interface.

1.1.1 mindspeed_ms.core.config.AllConfig

class mindspeed_ms.core.config.**AllConfig**(***kwargs*)

A Config that contains all other configs, which will be used in init_configs methods as the default config.

Parameters

kwargs (*dict*) –Extra keyword configuration arguments.

Supported Platforms:

Ascend

Examples

```
>>> from mindspeed_ms.core.config import AllConfig
>>> all_config = [AllConfig]
```

1.1.2 mindspeed_ms.core.config.DatasetConfig

class mindspeed_ms.core.config.**DatasetConfig**(*dataset_dir: str = './dataset', shuffle: bool = False, batch_size: int = 1, micro_batch_num: int = 1, train_samples: int = 0, data_layout: str = 'BSH', eos_token_id: int = 0, **kwargs*)

Dataset config class.

Parameters

- **dataset_dir** (*str, optional*) –Dataset file directory. Default: `"./dataset"`.
- **shuffle** (*bool, optional*) –Shuffle dataset. Default: `False`.
- **batch_size** (*int, optional*) –The number of data samples processed in one iteration. Default: `1`.
- **micro_batch_num** (*int, optional*) –Number of micro batch when using pipeline parallel or gradient accumulation. Default: `1`.
- **train_samples** (*int, optional*) –Number of train samples. Default: `0`.
- **data_layout** (*str, optional*) –Input layout. Default: `"BSH"`.
- **eos_token_id** (*int, optional*) –EoS token id. Default: `0`.
- **kwargs** (*dict*) –Extra keyword configuration arguments.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspeed_ms.core.config import DatasetConfig
>>> dataset_config = DatasetConfig()
```

1.1.3 mindspeed_ms.core.config.init_configs_from_yaml

mindspeed_ms.core.config.**init_configs_from_yaml**(*file_path: str, config_classes=None, **kwargs*)

Initialize config class from configuration yaml file.

Parameters

- **file_path** (*str*) –Configuration yaml file.
- **config_classes** (*Union[list[BaseConfig], None], optional*) –Config classes to be initialized. Support [TrainingConfig, ModelParallelConfig, OptimizerConfig, DatasetConfig, LoraConfig, TransformerConfig, MoEConfig]. When no config class is passed in, all known configs will be initialized as optional config of *mindspeed_ms.core.config.AllConfig*. Default: `None`.
- **kwargs** (*dict*) –Extra keyword configuration arguments.

Returns

Return initialized config instances, when no config class is passed in, *AllConfig* will be returned.

Raises

- **ValueError** –If *file_path* is not a string.
- **ValueError** –If the suffix of *file_path* does not end with yaml or yml.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspeed_ms.core.config import init_configs_from_yaml
>>> config_file = "/path/to/config/file"
>>> all_config = init_configs_from_yaml(config_file)
```

1.1.4 mindspeed_ms.core.config.ModelParallelConfig

class `mindspeed_ms.core.config.`**`ModelParallelConfig`**(*tensor_model_parallel_size: int = 1, pipeline_model_parallel_size: int = 1, context_parallel_size: int = 1, context_parallel_algo: str = 'ulysses_cp_algo', ulysses_degree_in_cp: int = None, expert_model_parallel_size: int = 1, virtual_pipeline_model_parallel_size: int = None, sequence_parallel: bool = False, recv_dtype: str = 'float32', zero_level: str = None, standalone_embedding_stage: bool = False, overlap_grad_reduce: bool = False, gradient_accumulation_fusion: bool = False, overlap_p2p_comm: bool = True, use_cpu_initialization: bool = False, deterministic_mode: bool = False, num_layer_list: list = None, recompute_config: dict = None, recompute: str = None, select_recompute: str = None, select_comm_recompute: str = None, variable_seq_lengths: bool = False, **kwargs*)

Parallel config class.

Parameters

- **`tensor_model_parallel_size`** (*`int, optional`*) –Dimensionality of tensor parallel. Default: `1`.
- **`pipeline_model_parallel_size`** (*`int, optional`*) –Number of stages when using pipeline parallel. Default: `1`.
- **`context_parallel_size`** (*`int, optional`*) –Dimensionality of context parallel. Default: `1`.
- **`context_parallel_algo`** (*`str, optional`*) –Context parallelism algorithm.

  Choices: [`"ulysses_cp_algo"`, `"megatron_cp_algo"`, `"hybrid_cp_algo"`].

  Default: `"ulysses_cp_algo"`.
- **`ulysses_degree_in_cp`** (*`int, optional`*) –Define the degree of ulysses parallelism when the `--context-parallel-algo` is set to `hybrid_cp_algo`, and the ring-attention parallelism is set to `cp//ulysses`. Default: `None`.
- **`expert_model_parallel_size`** (*`int, optional`*) –Dimensionality of expert parallel. Default: `1`.
- **`virtual_pipeline_model_parallel_size`** (*`int, optional`*) –Number of virtual stages when using pipeline parallel. Default: `None`.
- **`sequence_parallel`** (*`bool, optional`*) –Enable sequence parallel. Default: `False`.

- **`recv_dtype`** (*`str`*, *`optional`*) –Communication data type of p2p communication when using pipeline parallel. Default: `"float32"`.
- **`zero_level`** (*`str`*, *`optional`*) –Zero level for ZeRO optimizer, if `None`, will not use ZeRO optimizer. Default: `None`.
- **`standalone_embedding_stage`** (*`bool`*, *`optional`*) –Enable standalone embedding stage. Default: `False`.
- **`overlap_grad_reduce`** (*`bool`*, *`optional`*) –Enable overlap grad reduce. Default: `False`.
- **`gradient_accumulation_fusion`** (*`bool`*, *`optional`*) –Enable gradient accumulation during linear backward execution. Default: `False`.
- **`overlap_p2p_comm`** (*`bool`*, *`optional`*) –Enable overlap p2p commucation in pipeline interleaved. Default: `True`.
- **`use_cpu_initialization`** (*`bool`*, *`optional`*) –Use cpu initialization. Default: `False`.
- **`deterministic_mode`** (*`bool`*, *`optional`*) –Deterministic mode. Default: `False`.
- **`num_layer_list`** (*`list`*, *`optional`*) –User-defined pipeline parallel model layer division. Default: `None`.
- **`recompute_config`** (*`dict`*, *`optional`*) –Recompute config. Default: `None`.
- **`recompute`** (*`str`*, *`optional`*) –Enable recompute. Default: `None`
- **`select_recompute`** (*`str`*, *`optional`*) –Enable select recompute. Default: `None`
- **`select_comm_recompute`** (*`str`*, *`optional`*) –Enable select commucation recompute. Default: `None`
- **`variable_seq_lengths`** (*`bool`*, *`optional`*) –Enable variable sequence lengths. Default: `False`.
- **`kwargs`** (*`dict`*) –Extra keyword configuration arguments.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspeed_ms.core.config import ModelParallelConfig
>>> parallel_config = ModelParallelConfig()
```

1.1.5 mindspeed_ms.core.config.MoEConfig

class `mindspeed_ms.core.config.`**`MoEConfig`**(*num_experts: int = 1, moe_grouped_gemm: bool = False, moe_router_topk: int = 2, moe_router_load_balancing_type: str = 'none', moe_token_dispatcher_type: str = 'alltoall', use_self_defined_alltoall: bool = False, moe_expert_capacity_factor: float = None, moe_pad_expert_input_to_capacity: bool = False, moe_token_drop_policy: str = 'probs', moe_aux_loss_coeff: float = 0.0, moe_z_loss_coeff: float = None, moe_input_jitter_eps: float = None, **kwargs*)

MoE config class.

Parameters

- **`num_experts`** (*`int`*, *`optional`*) –The number of experts. Default: `1`.
- **`moe_grouped_gemm`** (*`bool`*, *`optional`*) –Use grouped gemm or not. Default: `False`.

- **moe_router_topk** (*int, optional*) –Router TopK number. Default: `2`.
- **moe_router_load_balancing_type** (*str, optional*) –Type of moe router load balancing algorithm. Choose from: [`"aux_loss"`, `"none"`]. Default: `"none"`.
- **moe_token_dispatcher_type** (*str, optional*) –Type of moe token dispatcher algorithm. Choose from: [`'alltoall'`]. Default: `'alltoall'`.
- **use_self_defined_alltoall** (*bool, optional*) –Use self-defined *alltoall* operators. Default: `False`.
- **moe_expert_capacity_factor** (*float, optional*) –The capacity factor for each expert. Default: `None`.
- **moe_pad_expert_input_to_capacity** (*bool, optional*) –Whether pads the input for each expert to match the expert capacity length. Default: `False`.
- **moe_token_drop_policy** (*str, optional*) –The policy to drop tokens. Default: `"probs"`.
- **moe_aux_loss_coeff** (*float, optional*) –Scaling coefficient for the aux loss. Default: `0.0`.
- **moe_z_loss_coeff** (*float, optional*) –Scaling coefficient for the z-loss. Default: `None`.
- **moe_input_jitter_eps** (*float, optional*) –Add noise to the input tensor by applying jitter with a specified epsilon value. Default: `None`.
- **kwargs** (*dict*) –Extra keyword configuration arguments.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspeed_ms.core.config import MoEConfig
>>> moe_config = MoEConfig(num_experts=4, moe_router_topk=2)
```

1.1.6 mindspeed_ms.core.config.TrainingConfig

class mindspeed_ms.core.config.**TrainingConfig**(*parallel_config: ModelParallelConfig, dataset_config: DatasetConfig = DatasetConfig(), lora_config: LoraConfig = LoraConfig(), seed: int = None, output_dir: str = './output', training_iters: int = 0, epochs: int = None, log_interval: int = None, eval_interval: int = None, save_interval: int = None, best_metric_comparison: str = None, eval_metric: str = None, grad_clip_kwargs: dict = None, loss_scale: Union[float, int] = None, loss_scale_value: Union[float, int] = None, loss_scale_factor: int = None, loss_scale_window: int = None, loss_reduction: str = 'mean', calculate_per_token_loss: bool = False, wrap_with_ddp: bool = False, accumulate_allreduce_grads_in_fp32: bool = False, overlap_grad_reduce: bool = False, delay_grad_reduce: bool = False, use_distributed_optimizer: bool = False, bucket_size: Optional[int] = None, check_for_nan_in_grad: bool = False, fp16: bool = False, bf16: bool = False, resume_training: bool = False, crc_check: bool = False, load_checkpoint: str = '', enable_compile_cache: bool = False, compile_cache_path: str = None, ckpt_format: str = 'ckpt', prefix: str = 'network', keep_checkpoint_max: int = 5, no_load_optim: bool = False, no_load_rng: bool = True, new_dataset: bool = False, enable_mem_align: bool = False, profile: bool = False, profile_save_path: str = None, profile_step_start: int = 1, profile_step_end: int = 5, profile_level: str = 'level0', profile_with_stack: bool = False, profile_memory: bool = False, profile_framework: str = 'all', profile_communication: bool = False, profile_parallel_strategy: bool = False, profile_aicore_metrics: int = 0, profile_l2_cache: bool = False, profile_hbm_ddr: bool = False, profile_pcie: bool = False, profile_data_process: bool = False, profile_data_simplification: bool = False, profile_op_time: bool = True, profile_offline_analyse: bool = False, profile_dynamic_profiler_config_path: str = '', **kwargs*)

Training config.

Parameters

- **parallel_config** (`ModelParallelConfig`) –Parallel config.
- **dataset_config** (`DatasetConfig`) –Dataset config. Default: `DatasetConfig()`.
- **lora_config** (*`LoraConfig`*) –Lora config. Default: `LoraConfig()`.
- **seed** (*`int, optional`*) –Random seed for initialization. Default: `None`.
- **output_dir** (*`str, optional`*) –Output directory for saving checkpoints, logs and so on. Default: `"./output"`.
- **training_iters** (*`int, optional`*) –Training iterations for training. Default: `0`.
- **epochs** (*`int, optional`*) –Epochs for training. Default: `None`.
- **log_interval** (*`int, optional`*) –Log interval for training. Default: `None`.
- **eval_interval** (*`int, optional`*) –Evaluation interval for training. Default: `None`.
- **save_interval** (*`int, optional`*) –Save interval for training. Default: `None`.

- **best_metric_comparison** (*str, optional*) –The method to compare best metric. Default: `None`.
- **eval_metric** (*str, optional*) –The name of evaluation metrics. Default: `None`.
- **grad_clip_kwargs** (*dict, optional*) –Gradient clip arguments. Default: `None`.
- **loss_scale** (*Union[float, int], optional*) –Initial value of loss scale. If set, will use static loss scaler. Default: `None`.
- **loss_scale_value** (*Union[float, int], optional*) –Initial value of dynamic loss scale. Default: `None`.
- **loss_scale_factor** (*int, optional*) –Factor of dynamic loss scale. Default: `None`.
- **loss_scale_window** (*int, optional*) –Window size of dynamic loss scale. Default: `None`.
- **loss_reduction** (*str, optional*) –Loss reduction method. Default: `"mean"`.
- **calculate_per_token_loss** (*bool, optional*) –Apply grad and loss calculation base on num of tokens. Default: `False`.
- **wrap_with_ddp** (*bool, optional*) –Using DistributedDataParallel to wrap model. Default: `False`.
- **accumulate_allreduce_grads_in_fp32** (*bool, optional*) –When fp32 is set `True`, whether to accumulate allreduce grads. Default: `False`.
- **overlap_grad_reduce** (*bool, optional*) –Enable gradient computing and synchronization communication overlap when using DistributedDataParallel. Default: `False`.
- **delay_grad_reduce** (*bool, optional*) –If set `True`, delay grad reductions in all but first PP stage. Default: `False`.
- **use_distributed_optimizer** (*bool, optional*) –Enable DistributedOptimizer when using DistributedDataParallel. Default: `False`.
- **bucket_size** (*Optional[int], optional*) –Bucket size which is used to partition buffer into buckets when *overlap_grad_reduce* is `True`. Default: `None`.
- **check_for_nan_in_grad** (*bool, optional*) –If set `True`, check gradients in buffer are finite after synchronization. Default: `False`.
- **fp16** (*bool, optional*) –Whether to use fp16 type. Default: `False`.
- **bf16** (*bool, optional*) –Whether to use bf16 type. Default: `False`.
- **resume_training** (*bool, optional*) –Resume training. Default: `False`.
- **crc_check** (*bool, optional*) –CRC check when save/load checkpoint. Enable this may cause low train performance. Default: `False`.
- **load_checkpoint** (*str, optional*) –Where to load checkpoint. Default: `""`.
- **enable_compile_cache** (*bool, optional*) –Save compile cache. Enable this may cause low train performance. Default: `False`.
- **compile_cache_path** (*str, optional*) –Where to save compile cache. Default: `None`.
- **ckpt_format** (*str, optional*) –Checkpoint save format. Default: `"ckpt"`.
- **prefix** (*str, optional*) –Checkpoint save prefix. Default: `"network"`.
- **keep_checkpoint_max** (*int, optional*) –Max saved checkpoint number. Default: `5`.
- **no_load_optim** (*bool, optional*) –When resume training, whether to load optimizer state. Default: `False`.
- **no_load_rng** (*bool, optional*) –When resume training, whether to load RNG state. Default: `True`.
- **new_dataset** (*bool, optional*) –When resume training, whether to use new dataset. Default: `False`.

- **enable_mem_align** (*bool, optional*) –Whether to enable memory align. Default: `False`.
- **profile** (*bool, optional*) –Whether to open profiling. Default: `False`.
- **profile_save_path** (*str, optional*) –Path to save profiling files. Default: `None`.
- **profile_step_start** (*int, optional*) –Profiling start step. Default: `1`.
- **profile_step_end** (*int, optional*) –Profiling end step. Default: `5`.
- **profile_level** (*str, optional*) –Profiling level. Default: `"level0"`.
- **profile_with_stack** (*bool, optional*) –Profiling with stack info. Default: `False`.
- **profile_memory** (*bool, optional*) –Profiling with memory info. Default: `False`.
- **profile_framework** (*str, optional*) –Profiling with framework info. Default: `"all"`.
- **profile_communication** (*bool, optional*) –Profiling with communication info. Default: `False`.
- **profile_parallel_strategy** (*bool, optional*) –Profiling with parallel strategy info. Default: `False`.
- **profile_aicore_metrics** (*int, optional*) –Profiling with aicore metrics info. Default: `0`.
- **profile_l2_cache** (*bool, optional*) –Profiling with l2 cache info. Default: `False`.
- **profile_hbm_ddr** (*bool, optional*) –Profiling with hbm ddr info. Default: `False`.
- **profile_pcie** (*bool, optional*) –Profiling with pcie info. Default: `False`.
- **profile_data_process** (*bool, optional*) –Profiling with data process info. Default: `False`.
- **profile_data_simplification** (*bool, optional*) –Profiling with data simplification. Default: `False`.
- **profile_op_time** (*bool, optional*) –Profiling with op time info. Default: `True`.
- **profile_offline_analyse** (*bool, optional*) –Profiling with offline analyse. Default: `False`.
- **profile_dynamic_profiler_config_path** (*str, optional*) –Path of dynamic profiler. Default: `""`.
- **kwargs** (*dict*) –Extra keyword configuration arguments.

Raises

ValueError –*fp16* is `True` and *bf16* is `True`.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspeed_ms.core.config import TrainingConfig, ModelParallelConfig
>>> parallel_config = ModelParallelConfig()
>>> training_config = TrainingConfig(parallel_config=parallel_config)
```

1.1.7 mindspeed_ms.core.config.TransformerConfig

class mindspeed_ms.core.config.**TransformerConfig**(*vocab_size: int, num_layers: int, num_attention_heads: int, hidden_size: int, ffn_hidden_size: int, parallel_config: ModelParallelConfig, training_config: TrainingConfig, lora_config: LoraConfig = LoraConfig(), dataset_config: DatasetConfig = DatasetConfig(), moe_config: MoEConfig = MoEConfig(), attention_type: str = 'self_attn', position_embedding_type: str = 'absolute', parallel_position_embedding: bool = False, rotary_config: dict = None, use_query_layer: bool = False, use_visual_encoder: bool = False, use_retriever: bool = False, group_query_attention: bool = False, num_query_groups: int = 32, qkv_has_bias: bool = True, out_proj_has_bias: bool = True, head_skip_weight_param_allocation: bool = True, apply_query_key_layer_scaling: bool = False, use_flash_attention: bool = False, fa_config=None, enable_flash_sp: bool = False, mask_func_type: str = 'attn_mask_add', mlp_has_bias: bool = True, hidden_act: str = 'gelu', normalization: str = 'LayerNorm', norm_epsilon: float = 1.0e-5, apply_residual_connection_post_norm: bool = False, use_final_norm: bool = True, residual_connection_dtype: str = 'float32', init_method_std: float = 0.01, params_dtype: str = 'float32', embedding_init_dtype: str = 'float32', compute_dtype: str = 'float32', softmax_compute_dtype: str = 'float32', init_method: str = 'normal', bias_init: str = 'zeros', fp16_lm_cross_entropy: bool = False, attention_dropout: float = 0.0, out_hidden_size: int = None, num_experts: int = None, untie_embeddings_and_output_weights: bool = False, flatten_labels_and_input_mask: bool = True, recompute_method: str = None, recompute_num_layers: int = None, recompute_granularity: str = None, fp32_residual_connection: bool = False, kv_channels: int = None, hidden_dropout: float = 0.0, bias_dropout_fusion: bool = False, fp8_format: str = None, clone_scatter_output_in_embedding: bool = False, add_bias_linear: bool = False, attention_softmax_in_fp32: bool = True, masked_softmax_fusion: bool = False, distribute_saved_activations: bool = False, retro_add_retriever: bool = False, transformer_impl: str = 'local', encoder_num_layers: int = None, decoder_num_layers: int = None, model_type: str = 'encoder_or_decoder', select_comm_recompute: bool = False, select_recompute: bool = False, apply_rope_fusion: bool = False, use_sandwich_norm: bool = False, attn_post_norm_scale: float = 1.0, ffn_post_norm_scale: float = 1.0, apply_swiglu_fusion: bool = False, **kwargs*)

Transformer config class.

Parameters

- **vocab_size** (*int*) –Vocabulary size.

- **num_layers** (*int*) –Number of model layers.
- **num_attention_heads** (*int*) –Number of heads for MultiHeadAttention.
- **hidden_size** (*int*) –Hidden size.
- **ffn_hidden_size** (*int*) –Hidden size of Feed-Forward Network.
- **parallel_config** (`ModelParallelConfig`) –Parallel config.
- **training_config** (`TrainingConfig`) –Training config.
- **lora_config** (*LoraConfig, optional*) –Lora config. Default: `LoraConfig()`.
- **dataset_config** (`DatasetConfig`, *optional*) –Dataset config. Default: `DatasetConfig()`.
- **moe_config** (`MoEConfig`, *optional*) –MoE config. Default: `MoEConfig()`.
- **attention_type** (*str, optional*) –Attention type. Default: `"self_attn"`.
- **position_embedding_type** (*str, optional*) –Position embedding type. Default: `'absolute'`.
- **parallel_position_embedding** (*bool, optional*) –Apply parallel vocab embedding layer when using absolute position embedding. Default: `False`.
- **rotary_config** (*dict, optional*) –Rotary position embedding config. Default: `None`.
- **use_query_layer** (*bool, optional*) –Whether to use a separate query layer. Default: `False`.
- **use_visual_encoder** (*bool, optional*) –Using visual encoder. Default: `False`.
- **use_retriever** (*bool, optional*) –Using retriever. Default: `False`.
- **group_query_attention** (*bool, optional*) –Enable group query attention. Default: `False`.
- **num_query_groups** (*int, optional*) –Number of heads for key and value when using group query attention. Default: `32`.
- **qkv_has_bias** (*bool, optional*) –Linears apply on query, key and value in Attention block has bias parameter. Default: `True`.
- **out_proj_has_bias** (*bool, optional*) –Linear applies on output of core attention block has bias parameter. Default: `True`.
- **head_skip_weight_param_allocation** (*bool, optional*) –If `True`, the Head will skip weight allocation and use word as weights. Default: `True`.
- **apply_query_key_layer_scaling** (*bool, optional*) –Apply query key scaling in core attention block. Default: `False`.
- **use_flash_attention** (*bool, optional*) –Enable flash attention. Default: `False`.
- **fa_config** (*dict, optional*) –Flash attention config. Default: `None`.
- **enable_flash_sp** (*bool, optional*) –Enable flash sp. Default: `False`.
- **mask_func_type** (*str, optional*) –Attention mask compute method. Default: `"attn_mask_add"`.
- **mlp_has_bias** (*bool, optional*) –Linears in MLP block have bias parameters. Default: `True`.
- **hidden_act** (*str, optional*) –Activation used in MLP block. Default: `"gelu"`.
- **normalization** (*str, optional*) –Normalization used in transformer layer block. Default: `"LayerNorm"`.
- **norm_epsilon** (*float, optional*) –Epsilon of normalization. Default: `1.0e-5`.
- **apply_residual_connection_post_norm** (*bool, optional*) –Apply residual connection after normalization. Default: `False`.

- **use_final_norm** (*bool, optional*) –Apply final norm after transformer. Default: `True`.
- **residual_connection_dtype** (*str, optional*) –Compute data type of residual connection. Default: `"float32"`.
- **init_method_std** (*float, optional*) –Init method std value. Default: `0.01`.
- **params_dtype** (*str, optional*) –Parameter initialize data type. Default: `"float32"`.
- **embedding_init_dtype** (*str, optional*) –Embedding parameter initialize data type. Default: `"float32 "`.
- **compute_dtype** (*str, optional*) –Compute data type of linear module. Default: `"float32"`.
- **softmax_compute_dtype** (*str, optional*) –Compute data type of softmax layer. Default: `"float32"`.
- **init_method** (*str, optional*) –Init method. Default: `'normal'`.
- **bias_init** (*str, optional*) –Bias init method. Default: `'zeros'`.
- **fp16_lm_cross_entropy** (*bool, optional*) –Apply float16 when calculating cross entropy. Default: `False`.
- **attention_dropout** (*float, optional*) –Dropout rate for attention module. Default: `0.0`.
- **out_hidden_size** (*int, optional*) –Out hidden size. Default: `None`.
- **num_experts** (*int, optional*) –Number of experts. Default: `None`.
- **untie_embeddings_and_output_weights** (*bool, optional*) –If `False`, share embedding with head layer. Default: `False`.
- **flatten_labels_and_input_mask** (*bool, optional*) –flatten labels and input mask. Default: `True`.
- **recompute_method** (*str, optional*) –Recompute method. Default: `None`.
- **recompute_num_layers** (*int, optional*) –Number of layers to recompute. Default: `None`.
- **recompute_granularity** (*str, optional*) –Recompute granularity. Default: `None`.
- **fp32_residual_connection** (*bool, optional*) –Enable fp32 residual connection. Default: `False`.
- **kv_channels** (*int, optional*) –Key and value channels. Default: `None`.
- **hidden_dropout** (*float, optional*) –Dropout rate for output of attention block and mlp block. Default: `0.0`.
- **bias_dropout_fusion** (*bool, optional*) –Enable bias dropout fusion. Default: `False`.
- **fp8_format** (*str, optional*) –Use fp8 format. Default: `None`.
- **clone_scatter_output_in_embedding** (*bool, optional*) –Enable clone scatter output in embedding. Default: `False`.
- **add_bias_linear** (*bool, optional*) –Enable bias linear. Default: `False`.
- **attention_softmax_in_fp32** (*bool, optional*) –Enable attention softmax in fp32. Default: `True`.
- **masked_softmax_fusion** (*bool, optional*) –Enable masked softmax fusion. Default: : `False`,
- **distribute_saved_activations** (*bool, optional*) –Enable distribute saved activations. Default: `False`.
- **retro_add_retriever** (*bool, optional*) –Enable retro add retriever. Default: `False`.
- **transformer_impl** (*str, optional*) –Transformer implementation. Default: `'local'`.
- **encoder_num_layers** (*int, optional*) –Encoder num layers. Default: `None`.

- **decoder_num_layers** (*int, optional*) –Decoder num layers. Default: `None`.
- **model_type** (*str, optional*) –Model type. Default: `"encoder_or_decoder"`.
- **select_comm_recompute** (*bool, optional*) –Enable select comm recompute. Default: `False`.
- **select_recompute** (*bool, optional*) –Enable select recompute. Default: `False`.
- **apply_rope_fusion** (*bool, optional*) –Enable rope fusion. Default: `False`.
- **use_sandwich_norm** (*bool, optional*) –Enable sandwich norm. Default: `False`.
- **attn_post_norm_scale** (*float, optional*) –Attention post norm scale. Default: `1.0`.
- **ffn_post_norm_scale** (*float, optional*) –Ffn post norm scale. Default: `1.0`.
- **apply_swiglu_fusion** (*bool, optional*) –Enable swiglu fusion. Default: `False`.
- **kwargs** (*dict*) –Extra keyword configuration arguments.

1.1.8 mindspeed_ms.core.parallel_state.get_context_parallel_rank

`mindspeed_ms.core.parallel_state.`**`get_context_parallel_rank`**`()`

Return rank for the context parallel group.

Returns

- The rank id.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables. For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.core.parallel_state import get_context_parallel_rank
>>> init()
>>> initialize_model_parallel(context_parallel_size=1)
>>> cp_rank = get_context_parallel_rank()
>>> print(cp_rank)
0
```

1.1.9 mindspeed_ms.core.parallel_state.get_context_parallel_world_size

`mindspeed_ms.core.parallel_state.`**`get_context_parallel_world_size`**`()`

Return world size for the context parallel group.

Returns

- World size.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables. For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.core.parallel_state import get_context_parallel_world_size
>>> init()
>>> initialize_model_parallel(context_parallel_size=1)
>>> cp_world_size = get_context_parallel_world_size()
>>> print(cp_world_size)
1
```

1.1.10 mindspeed_ms.core.parallel_state.initialize_model_parallel

`mindspeed_ms.core.parallel_state.`**`initialize_model_parallel`**`(`*tensor_model_parallel_size=1, pipeline_model_parallel_size=1, virtual_pipeline_model_parallel_size=None, pipeline_model_parallel_split_rank=None, context_parallel_size=1, expert_model_parallel_size=1, order='tp-cp-ep-dp-pp', communicator_config_path=None, **kwargs*`)`

Initialize model data parallel groups.

Parameters

- **`tensor_model_parallel_size`** (*`int`*, *`optional`*) –The number of devices to split individual tensors across. Default: `1`.
- **`pipeline_model_parallel_size`** (*`int`*, *`optional`*) –The number of tensor parallel device groups to split the Transformer layers across. Default: `1`.
- **`virtual_pipeline_model_parallel_size`** (*`int`*, *`optional`*) –The number of stages that each pipeline group will have, interleaving as necessary. If `None`, no interleaving is performed. Default: `None`.

- **pipeline_model_parallel_split_rank** (*int, optional*) –For models with both an encoder and decoder, the rank in pipeline to switch between encoder and decoder (i.e. the first rank of the decoder). This allows the user to set the pipeline parallel size of the encoder and decoder independently. Default: `None`.
- **context_parallel_size** (*int, optional*) –The number of tensor parallel device groups to split the network input sequence length across. Compute of attention module requires tokens of full sequence length, so devices in a context parallel group need to communicate with each other to exchange information of other sequence chunks. Each device and its counterparts in other tensor parallel groups compose a context parallel group. Default: `1`.
- **expert_model_parallel_size** (*int, optional*) –The number of devices to split experts across in MoE model. Default: `1`.
- **order** (*str, optional*) –The order each parallel strategy follows. Default: `"tp-cp-ep-dp-pp"`.
- **communicator_config_path** (*str, optional*) –Path to the yaml file of HCCL communicator configurations. Currently not in use. Default: `None`.
- **kwargs** (*dict*) –Extra keyword configuration arguments.

Raises

- `RuntimeError` –If *mindspore.communication._comm_helper* is not initialized.
- `RuntimeError` –If *world_size* is not divisible by (*tensor_model_parallel_size* * *pipeline_model_parallel_size* * *context_parallel_size*).
- `RuntimeError` –If *data_parallel_size* is not divisible by *expert_model_parallel_size* .
- `RuntimeError` –If *expert_model_parallel_size* > *1* and *context_parallel_size* > *1*.
- `RuntimeError` –If *virtual_pipeline_model_parallel_size* is not `None` and *pipeline_model_parallel_size* < 2.
- `RuntimeError` –If *order* is `None`.
- `RuntimeError` –If *order* has duplicate elements.
- `RuntimeError` –If *ep* in *order* then if *ep-dp* not in *order* and *dp-ep* not in *order*.
- `RuntimeError` –If *_GLOBAL_STREAM* is not `None`.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables. For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.core.parallel_state import get_context_parallel_world_size
>>> init()
>>> initialize_model_parallel(context_parallel_size=1)
>>> context_parallel_world_size = get_context_parallel_world_size()
>>> print(context_parallel_world_size)
1
```

1.1.11 mindspeed_ms.core.tensor_parallel.ColumnParallelLinear

class mindspeed_ms.core.tensor_parallel.**ColumnParallelLinear**(*input_size*, *output_size*, ***, *config*, *init_method*, *bias=True*, *gather_output=False*, *stride=1*, *keep_master_weight_for_test=False*, *skip_bias_add=False*, *skip_weight_param_allocation=False*, *embedding_activation_buffer=None*, *grad_output_buffer=None*, *is_expert=False*, *tp_comm_buffer_name=None*, *disable_grad_reduce=False*, *bias_init=Zero()*, *param_init_dtype=None*, *compute_dtype=None*, *transpose_b=True*)

The dense layer with weight sliced on second dimension by tensor parallel size. This layer implements the operation as:

$$\text{outputs} = \text{inputs} * \text{weight} + \text{bias},$$

where $inputs$ is the input tensors, weight is a weight matrix created by the layer, and bias is a bias vector created by the layer (only if *bias* is True).

Parameters

- **input_size** (*int*) –The number of channels in the input space.
- **output_size** (*int*) –The number of channels in the output space.

Keyword Arguments

- **config** (*dict*) –The config of the transformer model. For details, please refer to TransformerConfig.
- **init_method** (*Union[Tensor, str, Initializer, numbers.Number]*) –The trainable weight_init parameter. The values of str refer to the function *initializer*.
- **bias** (*bool, optional*) –Specifies whether the layer uses a bias vector. Default: `True`.
- **gather_output** (*bool, optional*) –Specifies whether gather the output on each tensor parallel rank. Default: `False`.
- **stride** (*int, optional*) –For the strided linear layers. Default: `1`.
- **keep_master_weight_for_test** (*bool, optional*) –For testing and should be set to False. It returns the master weights used for initialization. Default: `False`.
- **skip_bias_add** (*bool, optional*) –If True, do not add the bias term, instead return it for fusion. Default: `False`.
- **skip_weight_param_allocation** (*bool, optional*) –Specifies whether skip the initialization of weight parameter. When set True, an weight tensor should be passed to construct function. Default: `False`.
- **embedding_activation_buffer** (*Tensor, optional*) –This buffer holds the input activations of the final embedding linear layer on the last pipeline stage. Default: `None`.
- **grad_output_buffer** (*Tensor, optional*) –This buffer holds the gradient outputs of the final embedding linear layer on the last pipeline stage. Default: `None`.
- **is_expert** (*bool, optional*) –Specifies whether this linear layer is an expert. Default: `False`.

- **tp_comm_buffer_name** (*str, optional*) –Communication buffer name is not used in non-Transformer-Engine modules. Default: `None`.
- **disable_grad_reduce** (*bool, optional*) –If True, reduction of output gradients across tensor-parallel ranks will be disabled. Default: `False`.
- **bias_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –The trainable bias_init parameter. The values of str refer to the function *initializer*. Default: `Zero()`.
- **param_init_dtype** (*dtype.Number, optional*) –The parameter initialization type. Default: `None`.
- **compute_dtype** (*dtype.Number, optional*) –The computation type. Default: `None`.
- **transpose_b** (*bool, optional*) –Specifies whether the weight parameter will be initialized as a transposed shape. Default: `True`.

Inputs:

- **input_** (Tensor) - Tensor of shape $(*, in\_channels)$. The *input_size* in *Args* should be equal to $in\_channels$ in *Inputs*.
- **weight** (Tensor) - Tensor of shape $(in\_channels, out\_channels)$. The *input_size* in *Args* should be equal to $in\_channels$ in *Inputs*. The *output_size* in *Args* should be equal to $out\_channels$ in *Outputs*.

Outputs:

- **output** (Tensor) - Tensor of shape $(*, out\_channels)$. The *output_size* in *Args* should be equal to $out\_channels$ in *Outputs*.
- **output_bias** (Tensor) - Tensor of shape $(out\_channels,)$.

Raises

- `ValueError` –*skip_weight_param_allocation=True* but weight_tensor is not passed to construct function.
- `NotImplementedError` –*stride > 1* is not supported for now.
- `NotImplementedError` –*keep_master_weight_for_test=True* is not supported for now.
- `NotImplementedError` –*embedding_activation_buffer* is not supported for now.
- `NotImplementedError` –*grad_output_buffer* is not supported for now.
- `NotImplementedError` –*tp_comm_buffer_name* is not supported for now.
- `NotImplementedError` –*disable_grad_reduce=True* is not supported for now.
- `NotImplementedError` –*config.parallel_config.use_cpu_initialization* is not supported for now.
- `RuntimeError` –use zero3 optimizer parallel without initializing data parallel communication.
- `RuntimeError` –*allreduce_dgrad* and *sequence_parallel* cannot be enabled at the same time.

Supported Platforms:

```
Ascend
```

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn, Tensor
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.core.config import (
...     ModelParallelConfig,
...     TrainingConfig,
...     TransformerConfig
... )
>>> from mindspeed_ms.core.tensor_parallel import ColumnParallelLinear
>>> class TestNet(nn.Cell):
...     def __init__(self, config):
...         super(TestNet, self).__init__()
...         hidden_size = config.hidden_size
...         self.columnlinear = ColumnParallelLinear(
...             input_size=hidden_size,
...             output_size=hidden_size,
...             config=config,
...             init_method=config.init_method,
...             bias=config.mlp_has_bias,
...             gather_output=False,
...             skip_bias_add=False,
...             bias_init=config.bias_init
...         )
...     def construct(self, input_):
...         output, _ = self.columnlinear(input_)
...         return output
>>> dataset_size = 1
>>> seq_length = 2
>>> hidden_size = 4
>>> tensor_parallel = 1
>>> ms.set_context(device_target='Ascend', mode=ms.PYNATIVE_MODE)
>>> ms.set_seed(2024)
>>> init()
>>> initialize_model_parallel(tensor_model_parallel_size=tensor_parallel)
>>> input_shape = (dataset_size, seq_length, hidden_size)
>>> input_data = Tensor(np.random.random(input_shape).astype(np.float32))
>>> parallel_config = ModelParallelConfig()
>>> train_config = TrainingConfig(parallel_config=parallel_config)
>>> model_config = TransformerConfig(vocab_size=40000,
...                                  num_layers=1,
...                                  num_attention_heads=1,
...                                  hidden_size=hidden_size,
...                                  ffn_hidden_size=4*hidden_size,
...                                  parallel_config=parallel_config,
...                                  training_config=train_config,
...                                  mlp_has_bias=True,
...                                  gated_linear_unit=False,
```

(continues on next page)

(continued from previous page)

```
...                                  hidden_act='gelu',
...                                  params_dtype='float32',
...                                  compute_dtype='float32')
>>> network = TestNet(config=model_config)
>>> output = network(input_data)
>>> print(output)
>>> print(output.shape)
[[[ 0.01780816  0.00895902 -0.00554341 -0.00185049]]
 [[ 0.02319741 -0.00320548 -0.0062025  -0.0050142 ]]]
(1, 2, 4)
```

1.1.12 mindspeed_ms.core.tensor_parallel.RowParallelLinear

class `mindspeed_ms.core.tensor_parallel.`**`RowParallelLinear`**(*input_size*, *output_size*, ***, *config*, *init_method*, *bias*, *input_is_parallel*, *skip_bias_add=True*, *stride=1*, *keep_master_weight_for_test=False*, *is_expert=False*, *tp_comm_buffer_name=None*, *bias_init=Zero()*, *param_init_dtype=None*, *compute_dtype=None*, *transpose_b=True*)

The dense layer with weight sliced on first dimension by tensor parallel size. This layer implements the operation as:

$$\text{outputs} = \text{inputs} * \text{weight} + \text{bias},$$

where $inputs$ is the input tensors, weight is a weight matrix created by the layer, and bias is a bias vector created by the layer (only if has_bias is True).

Parameters

- **`input_size`** (*`int`*) –The number of channels in the input space.
- **`output_size`** (*`int`*) –The number of channels in the output space.

Keyword Arguments

- **`config`** (*`dict`*) –The config of the transformer model. For details, please refer to TransformerConfig.
- **`init_method`** (*`Union[Tensor, str, Initializer, numbers.Number]`*) –The trainable weight_init parameter. The values of str refer to the function *initializer*.
- **`bias`** (*`bool`*) –Specifies whether the layer uses a bias vector.
- **`input_is_parallel`** (*`bool`*) –If True, we assume that the input is already split across the tensor parallel group and we do not split again.
- **`skip_bias_add`** (*`bool, optional`*) –If True, do not add the bias term, instead return it for fusion. Default: `True`.
- **`stride`** (*`int, optional`*) –For the strided linear layers. Default: `1`.
- **`keep_master_weight_for_test`** (*`bool, optional`*) –For tesing and should be set to False. It returns the master weights used for initialization. Default: `False`.
- **`is_expert`** (*`bool, optional`*) –Specifies whether this linear layer is an expert. Default: `False`.
- **`tp_comm_buffer_name`** (*`str, optional`*) –Communication buffer name is not used in non-Transformer-Engine modules. Default: `None`.

- **bias_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –The trainable bias_init parameter. The values of str refer to the function *initializer*. Default: `Zero()`.
- **param_init_dtype** (*dtype.Number, optional*) –The parameter initialization type. Default: `None`.
- **compute_dtype** (*dtype.Number, optional*) –The computation type. Default: `None`.
- **transpose_b** (*bool, optional*) –Specifies whether the weight parameter will be initialized as a transposed shape. Default: `True`.

Inputs:

- **input_** (Tensor) - Tensor of shape $(*, in\_channels)$. The *input_size* in *Args* should be equal to $in\_channels$ in *Inputs*.

Outputs:

- **output** (Tensor) - Tensor of shape $(*, out\_channels)$. The *output_size* in *Args* should be equal to $out\_channels$ in *Outputs*.
- **output_bias** (Tensor) - Tensor of shape $(out\_channels,)$.

Raises

- **ValueError** –*sequence_parallel* should be False when *input_is_parallel* is False , but got `True`.
- **ValueError** –*skip_bias_add* should be True when *explicit_expert_comm* is True , but got `False`.
- **NotImplementedError** –*stride > 1* is not supported for now.
- **NotImplementedError** –*keep_master_weight_for_test=True* is not supported for now.
- **NotImplementedError** –*tp_comm_buffer_name* is not supported for now.
- **RuntimeError** –Use zero3 optimizer parallel without initializing data parallel communication.
- **RuntimeError** –To enable *sequence_parallel*, *input_is_parallel* must be True but got `False`.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn, Tensor
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.core.config import (
...     ModelParallelConfig,
...     TrainingConfig,
...     TransformerConfig
... )
>>> from mindspeed_ms.core.tensor_parallel import RowParallelLinear
```

(continues on next page)

(continued from previous page)

```
>>> class TestNet(nn.Cell):
...     def __init__(self, config):
...         super(TestNet, self).__init__()
...         hidden_size = config.hidden_size
...         self.rowlinear = RowParallelLinear(input_size=hidden_size,
...                                            output_size=hidden_size,
...                                            config=config,
...                                            init_method=config.init_method,
...                                            bias=config.mlp_has_bias,
...                                            input_is_parallel=True,
...                                            skip_bias_add=False,
...                                            bias_init=config.bias_init)
...     def construct(self, input_):
...         output, _ = self.rowlinear(input_)
...         return output
>>> seq_length = 2
>>> dataset_size = 1
>>> hidden_size = 4
>>> tensor_parallel = 1
>>> ms.set_context(device_target='Ascend', mode=ms.PYNATIVE_MODE)
>>> ms.set_seed(2024)
>>> init()
>>> initialize_model_parallel(tensor_model_parallel_size=tensor_parallel)
>>> shape = (seq_length, dataset_size, hidden_size)
>>> input_data = Tensor(np.random.random(shape).astype(np.float32))
>>> parallel_config = ModelParallelConfig()
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> model_config = TransformerConfig(vocab_size=40000,
...                                   num_layers=1,
...                                   num_attention_heads=1,
...                                   hidden_size=hidden_size,
...                                   ffn_hidden_size=4*hidden_size,
...                                   parallel_config=parallel_config,
...                                   training_config=training_config,
...                                   mlp_has_bias=True,
...                                   gated_linear_unit=False,
...                                   hidden_act='gelu',
...                                   params_dtype='float32',
...                                   compute_dtype='float32')
>>> network = TestNet(config=model_config)
>>> output = network(input_data)
>>> print(output)
>>> print(output.shape)
[[[ 0.01780816  0.00895902 -0.00554341 -0.00185049]]
 [[ 0.02319741 -0.00320548 -0.0062025  -0.0050142 ]]]
(2, 1, 4)
```

1.1.13 mindspeed_ms.core.tensor_parallel.VocabParallelCrossEntropy

class `mindspeed_ms.core.tensor_parallel.`**`VocabParallelCrossEntropy`**(**args*, ***kwargs*)

Calculate the paralleled cross entropy loss.

Parameters

- **args** (*tuple*) –Positional arguments.
- **kwargs** (*dict*) –Other input.

Inputs:

- **vocab_parallel_logits** (Tensor) - Tensor of shape (N, C). Data type must be float16 or float32. The output logits of the backbone.
- **target** (Tensor) - Tensor of shape $(N,)$. The ground truth label of the sample.
- **label_smoothing** (float, optional) - smoothing factor, must be in range[0.0, 1.0). Default: `0.0`.

Outputs:

- **loss** (Tensor) - The corresponding cross entropy loss.

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.core.tensor_parallel.cross_entropy import         ...    ␣
↪VocabParallelCrossEntropy
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> init()
>>> initialize_model_parallel()
>>> loss = VocabParallelCrossEntropy()
>>> logits = Tensor([[2., 1., 0.2],[2., 1., 0.2]], mstype.float32)
>>> labels = Tensor([1,1], mstype.int32)
>>> output = loss(logits, labels)
>>> print(output.shape)
(2,)
>>> print(output)
[1.4273429 1.4273429]
```

1.1.14 mindspeed_ms.core.tensor_parallel.VocabParallelEmbedding

class mindspeed_ms.core.tensor_parallel.**VocabParallelEmbedding**(*num_embeddings*, *embedding_dim*, ***, *init_method*, *reduce_scatter_embeddings=False*, *config*, *param_init_dtype=None*)

Embedding parallelized in the vocabulary dimension.

Parameters

- **num_embeddings** (*int*) –vocabulary size.
- **embedding_dim** (*int*) –size of hidden state.

Keyword Arguments

- **init_method** (*Union[Tensor, str, Initializer, numbers.Number]*) –The trainable weight_init parameter. The values of str refer to the function *initializer*.
- **reduce_scatter_embeddings** (*bool, optional*) –Decides whether to perform ReduceScatter after embedding lookup. Default: `False`.
- **config** (*dict*) –The config of the transformer model. For details, please refer to TransformerConfig.
- **param_init_dtype** (*dtype.Number, optional*) –The parameter initialization type. Default: `None`.

Inputs:

- **input_** (Tensor) - Tensor of shape (B, S) or (S, B).

Outputs:

- **output** (Tensor) - Tensor of shape (B, S, H) or (S, B, H), which is consistent with the input.

Raises

- **ValueError** –The vocabulary size is not divisible by size of tensor parallel.
- **NotImplementedError** –*config.parallel_config.deterministic_mode* is not supported for now.
- **NotImplementedError** –*config.parallel_config.use_cpu_initialization* is not supported for now.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 4 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn, ops, Tensor
>>> from mindspore.communication.management import init
```

(continues on next page)

(continued from previous page)

```
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.core.tensor_parallel.layers import VocabParallelEmbedding
>>> from mindspeed_ms.core.config import (
...     ModelParallelConfig,
...     TrainingConfig,
...     DatasetConfig,
...     TransformerConfig
... )
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE, deterministic='ON')
>>> ms.set_context(pynative_synchronize=True)
>>> ms.set_seed(2024)
>>> ms.set_auto_parallel_context(parallel_mode=ms.ParallelMode.DATA_PARALLEL)
>>> class ParallelTransformerLayerNet(nn.Cell):
...     def __init__(self, config):
...         super(ParallelTransformerLayerNet, self).__init__()
...         self.config = config
...         self.embedding = VocabParallelEmbedding(
...             num_embeddings=config.vocab_size,
...             embedding_dim=config.hidden_size,
...             init_method=config.init_method,
...             reduce_scatter_embeddings=config.parallel_config.sequence_parallel,
...             config=config,
...         )
...     def construct(self, x):
...         x = self.embedding(x)
...         return x
>>> parallel_config = ModelParallelConfig(tensor_model_parallel_size=2,
...                                       pipeline_model_parallel_size=1,
...                                       context_parallel_size=1,
...                                       expert_model_parallel_size=1,
...                                       sequence_parallel=True)
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> dataset_config = DatasetConfig(batch_size=1,
...                                dataset_size=2,
...                                seq_length=1024)
>>> model_config = TransformerConfig(vocab_size=50304,
...                                  num_layers=1,
...                                  num_attention_heads=32,
...                                  hidden_size=2560,
...                                  ffn_hidden_size=7680,
...                                  parallel_config=parallel_config,
...                                  training_config=training_config)
>>> model_config.dataset_config = dataset_config
>>> batch_size = dataset_config.batch_size
>>> dataset_size = dataset_config.dataset_size
>>> seq_length = dataset_config.seq_length
>>> vocab_size = model_config.vocab_size
>>> init()
>>> tensor_parallel = parallel_config.tensor_model_parallel_size
>>> initialize_model_parallel(tensor_model_parallel_size=tensor_parallel)
>>> network = ParallelTransformerLayerNet(config=model_config)
>>> input_shape = (batch_size, seq_length)
>>> input_ids = Tensor(np.ones(input_shape).astype(np.int32))
>>> output = network(input_ids)
>>> print(output)
[[[-0.00546161  0.00440422  0.00252223 ...  0.00539334 -0.00625365 -0.01025379]
  [-0.00546161  0.00440422  0.00252223 ...  0.00539334 -0.00625365 -0.01025379]
```

(continues on next page)

(continued from previous page)

```
 [-0.00546161  0.00440422  0.00252223 ...  0.00539334 -0.00625365 -0.01025379]
 ...
 [-0.00546161  0.00440422  0.00252223 ...  0.00539334 -0.00625365 -0.01025379]
 [-0.00546161  0.00440422  0.00252223 ...  0.00539334 -0.00625365 -0.01025379]
 [-0.00546161  0.00440422  0.00252223 ...  0.00539334 -0.00625365 -0.01025379]]]
```

1.1.15 mindspeed_ms.legacy.model.eos_mask.EosMask

class `mindspeed_ms.legacy.model.eos_mask.`**`EosMask`**(*batch_size*, *seq_len*, *eod_token_id*, *reset_position_ids*)

Generate attention mask corresponding to a specific token.

Parameters

- **batch_size** (*int*) –Batch size.
- **seq_len** (*int*) –Sequence length.
- **eod_token_id** (*int*) –End-of-Document token id.
- **reset_position_ids** (*bool*) –If `True`, the position ids are reset.

Inputs:

- **input_ids** (Tensor) - Input indexes. Tensor of shape (B, S).

Outputs:

- **position_ids** (Tensor) - Position id. Tensor of shape (B, S).
- **mint.sub(1, mask)** (Tensor) - Mask. Tensor of shape (B, S, S).

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more detailgit stats.

```
>>> from mindspore.communication.management import init
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, nn
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.legacy.model.eos_mask import EosMask
>>> ms.set_context(device_target='Ascend', mode=ms.PYNATIVE_MODE)
>>> ms.set_seed(2024)
>>> init()
>>> initialize_model_parallel()
>>> b = 2
>>> s = 4
>>> eod_token_id = 4
>>> loss = EosMask(b, s, eod_token_id, reset_position_ids=False)
>>> input_ids = ms.Tensor(np.random.random((b, s)).astype(np.float32))
```

(continues on next page)

(continued from previous page)

```
>>> output, mask = loss(input_ids)
>>> print(output.shape)
>>> print(mask.shape)
(2, 4)
(2, 4, 4)
```

1.1.16 mindspeed_ms.legacy.model.gpt_model.GPTModel

class `mindspeed_ms.legacy.model.gpt_model.`**`GPTModel`**(*config*, *num_tokentypes=0*, *parallel_output=True*, *pre_process=True*, *post_process=True*, ***kwargs*)

The Generative Pre-trained Transformer (GPT) is a decoder-only Transformer model.

Parameters

- **config** (`TransformerConfig`) –The config of the transformer model. For details, please refer to TransformerConfig.
- **num_tokentypes** (*int, optional*) –size of the token-type embeddings. If > 0, using tokentypes embedding. Default: `0`.
- **parallel_output** (*bool, optional*) –Specifies whether return paralleled output on each tensor parallel rank. Default: `True`.
- **pre_process** (*bool, optional*) –When using pipeline parallel, indicate whether it's the first stage. Default: `True`.
- **post_process** (*bool, optional*) –When using pipeline parallel, indicate whether it's the last stage. Default: `True`.
- **kwargs** (*dict*) –Other input.

Inputs:

- **tokens** (Tensor) - Input indices. Shape (B, S).
- **position_ids** (Tensor) - Position offset. Shape (B, S).
- **attention_mask** (Tensor) - Attention mask. Shape (B, S).
- **loss_mask** (Tensor) - Loss mask. Shape (B, S).
- **retriever_input_ids** (Tensor, optional) - Retriever input token indices. Shape: Depends on the input shape of the retrieval task. Default: `None`.
- **retriever_position_ids** (Tensor, optional) - Retriever input position indices. Shape: Depends on the input shape of the retrieval task. Default: `None`.
- **labels** (Tensor, optional) - Tensor of shape $(N,)$. The ground truth label of the sample. Default: `None`.
- **tokentype_ids** (Tensor, optional) - List of token type ids to be fed to a model. Shape (B, S). Default: `None`.
- **inference_params** (Tensor, optional) - Inference parameters. Used to specify specific settings during inference, such as maximum generation length, max batch size, etc. Default: `None`.

Outputs:

- Returns gpt loss or hidden states.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import os
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.core.config import (
...     ModelParallelConfig,
...     TrainingConfig,
...     DatasetConfig,
...     TransformerConfig
... )
>>> from mindspeed_ms.legacy.model.gpt_model import GPTModel
>>> ms.set_context(device_target='Ascend', mode=ms.PYNATIVE_MODE)
>>> init()
>>> initialize_model_parallel()
>>> os.environ['HCCL_BUFFSIZE'] = "200"
>>> batch_size = 8
>>> seq_length = 32
>>> parallel_config = ModelParallelConfig()
>>> data_config = DatasetConfig(batch_size=batch_size)
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> config = TransformerConfig(vocab_size=128,
...                            seq_length=seq_length,
...                            num_layers=4,
...                            num_attention_heads=4,
...                            num_query_groups=32,
...                            hidden_size=64,
...                            ffn_hidden_size=256,
...                            parallel_config=parallel_config,
...                            training_config=training_config,
...                            dataset_config=data_config)
>>> gpt_model = GPTModel(config)
>>> input_data = Tensor(np.random.random((batch_size, seq_length)).astype(np.int32))
>>> attention_mask = Tensor(np.zeros((batch_size, 1, seq_length, seq_length)).astype(np.
↪int32))
>>> loss_mask = Tensor(np.random.random((batch_size, seq_length)).astype(np.int32))
>>> lm_output = gpt_model(tokens=input_data,
...                       position_ids=None,
...                       attention_mask=attention_mask,
...                       loss_mask=loss_mask)
>>> print(lm_output.shape)
(32, 8, 128)
```

1.1.17 mindspeed_ms.legacy.model.language_model.Embedding

class mindspeed_ms.legacy.model.language_model.**Embedding**(*hidden_size*, *vocab_size*, *max_sequence_length*, *embedding_dropout_prob*, *config*, *num_tokentypes=0*, ***kwargs*)

An embedding layer contain word embedding, position embedding and tokentypes embedding.

Parameters

- **hidden_size** (*int*) –Hidden states size for embedding layer.
- **vocab_size** (*int*) –Vocabulary size.
- **max_sequence_length** (*int*) –Maximum size of sequence. This is used for positional embedding. if using position embedding, it is necessary to set the maximum sequence length.
- **embedding_dropout_prob** (*float*) –Dropout rate for embedding layer.
- **config** (TransformerConfig) –The transformer configuration include init_method, parallel_config, etc.
- **num_tokentypes** (*int, optional*) –Size of the token-type embeddings. If > 0, using tokentypes embedding. Default: `0`.
- **kwargs** (*dict*) –Other input.

Inputs:

- **input_ids** (Tensor) - The tokenized inputs with datatype int32, shape (B, S).
- **position_ids** (Tensor) - Position ids for position embedding, shape (B, S).
- **tokentype_ids** (Tensor) - Token type IDs used to distinguish different types of tokens (e.g., sentence A and sentence B in BERT), with datatype int32, shape (B, S).

Outputs:

- **embeddings** (Tensor) - The embedding output, shape (B, S, H).

Raises

- **NotImplementedError** –If *config.clone_scatter_output_in_embedding* is True.
- **RuntimeError** –If *tokentype_ids* is not None and *tokentype_embeddings* is None. If *tokentype_ids* is None and *tokentype_embeddings* is not None.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore.communication import init
>>> from mindspeed_ms.core.config import TransformerConfig, ModelParallelConfig
>>> from mindspeed_ms.legacy.model.language_model import Embedding
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE, deterministic='ON')
>>> init()
>>> initialize_model_parallel()
>>> parallel_config = ModelParallelConfig()
>>> config = TransformerConfig(vocab_size=128,
...                            num_layers=1,
...                            num_attention_heads=8,
...                            num_query_groups=4,
...                            hidden_size=256,
...                            ffn_hidden_size=128,
...                            parallel_config=parallel_config,
...                            training_config=None)
>>> embedding = Embedding(hidden_size=config.hidden_size,
...                       vocab_size=128,
...                       max_sequence_length=64,
...                       embedding_dropout_prob=0.0,
...                       config=config)
>>> shape = (2, 64)
>>> input_ids = ms.Tensor(np.random.randint(0, 100, size=shape), dtype=ms.int32)
>>> position_array = np.expand_dims(np.arange(64), axis=0)
>>> position_array = position_array.repeat(repeats=2, axis=0)
>>> position_ids = ms.Tensor(position_array, dtype=ms.int32)
>>> out = embedding(input_ids, position_ids)
>>> print(out.shape)
(2, 64, 256)
```

1.1.18 mindspeed_ms.legacy.model.language_model.get_language_model

`mindspeed_ms.legacy.model.language_model.`**`get_language_model`**(*config*, *num_tokentypes*, *add_pooler*, *encoder_attn_mask_type*, *add_encoder=True*, *add_decoder=False*, *decoder_attn_mask_type=None*, *pre_process=True*, *post_process=True*)

Use this function to get language model.

Parameters

- **config** (`TransformerConfig`) –The config of the transformer model. For details, please refer to TransformerConfig.
- **num_tokentypes** (*int*) –If greater than `0`, using tokentypes embedding.
- **add_pooler** (*bool*) –If `True`, use pooler.
- **encoder_attn_mask_type** (*int*) –Encoder attention mask type.
- **add_encoder** (*bool, optional*) –If `True`, use encoder. Default: `True`.
- **add_decoder** (*bool, optional*) –If `True`, use decoder. Default: `False`.
- **decoder_attn_mask_type** (*int, optional*) –Decoder attention mask type. Default: `None`.

- **pre_process** (*bool, optional*) –When using pipeline parallel, indicate whether it's the first stage. Default: `True`.
- **post_process** (*bool, optional*) –When using pipeline parallel, indicate whether it's the last stage. Default: `True`.

Returns

- **language_model** (TransformerLanguageModel) - Transformer model.
- **language_model_key** (str) - Model key.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables. For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import os
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.communication import init
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.core.config import (
...     init_configs_from_yaml,
...     TrainingConfig,
...     ModelParallelConfig,
...     TransformerConfig,
...     DatasetConfig,
... )
>>> from mindspeed_ms.legacy.model.language_model import get_language_model
>>> os.environ['HCCL_BUFFSIZE'] = "1"
>>> config_path = "test_language_model.yaml"
>>> training_config, parallel_config, dataset_config, model_config = init_configs_from_yaml(
...     config_path, [TrainingConfig, ModelParallelConfig, DatasetConfig, TransformerConfig]
... )
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE,
...     deterministic="ON")
>>> init()
>>> initialize_model_parallel()
>>> batch_size = dataset_config.batch_size
>>> sq = model_config.seq_length
>>> language_model, _ = get_language_model(model_config,
...                                         num_tokentypes=0,
...                                         add_pooler=False,
...                                         encoder_attn_mask_type=None)
>>> input_data = Tensor(np.random.random((batch_size, sq)).astype(np.int32))
>>> attention_mask = Tensor(np.zeros((batch_size, 1, sq, sq)).astype(np.int32))
>>> hidden_states = language_model(input_data, None, attention_mask)
```

1.1.19 mindspeed_ms.legacy.model.module.Module

class mindspeed_ms.legacy.model.module.**Module**(*config=None*, *share_embeddings_and_output_weights=True*, ***kwargs*)

Specific extensions of cell with support for pipelining.

Parameters

- **config** (*dict, optional*) –The configuration of model. If it is not `None`, the *self.pre_process*, *self.post_process* will be set according to the pipeline stage. Default: `None`.
- **share_embeddings_and_output_weights** (*bool, optional*) –Decide whether to share the embeddings and output weights. If it is not `True`, *shared_embedding_or_output_weight()* and *initialize_word_embeddings()* could not be called. Default: `True`.
- **kwargs** (*dict*) –Extra keyword configuration arguments.

Raises

- **RuntimeError** –If more than one weight were set *'share'* attribute in a pipeline stage.
- **RuntimeError** –If there is one weight with *'share'* attribute in the model, but parameter sharing requires two weights with *'share'* attribute in first stage and last stage respectively.
- **RuntimeError** –If *share_embeddings_and_output_weights* is not `True`

  when *shared_embedding_or_output_weight()* is called.
- **RuntimeError** –If *share_embeddings_and_output_weights* is not `True`

  when *initialize_word_embeddings()* is called.
- **ValueError** –If it is the last stage (post process) but the sum of *shared_weight* is not `0.0` .

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspeed_ms.legacy.model.module import Module
>>> class CellExample(Module):
...     def __init__(self, layers):
...         super(CellExample, self).__init__()
...         self.layers = layers
...     def construct(self, hidden_states, *args, **kwargs):
...         for layer in self.layers:
...             hidden_states = layer(hidden_states, *args, **kwargs)
...         return hidden_states
```

1.1.20 mindspeed_ms.legacy.model.moe.experts.SequentialMLP

class mindspeed_ms.legacy.model.moe.experts.**SequentialMLP**(*num_local_experts: int*, *config: TransformerConfig*, *submodules=None*)

Define SequentialMLP module.

Parameters

- **num_local_experts** (*int*) –The number of local experts.
- **config** (TransformerConfig) –Configuration object for the transformer model.
- **submodules** (*MLPSubmodules*) –Type of the linear fully connected layer. Reserved parameter, currently not in use.

Inputs:

- **permuted_local_hidden_states** (Tensor) - The permuted input hidden states of the local experts.
- **token_per_expert** (Tensor) - The number of tokens per expert.

Outputs:

Tuple of 2 Tensor.

- **output_local** (Tensor) - The output of the local experts.
- **output_bias_local** (Tensor) - The bias of the local experts. Currently not in use. The return value is `None`.

Raises

NotImplementedError –If *submodules* is not None.

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore.communication import init
>>> from mindspeed_ms.core.config import TransformerConfig, ModelParallelConfig,␣
↪TrainingConfig
>>> from mindspeed_ms.legacy.model.moe.experts import SequentialMLP
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE, deterministic='ON')
>>> init()
>>> initialize_model_parallel()
>>> parallel_config = ModelParallelConfig()
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> config = TransformerConfig(vocab_size=128,
...                            num_layers=1,
...                            num_attention_heads=8,
...                            num_query_groups=4,
...                            hidden_size=64,
...                            ffn_hidden_size=128,
```

(continues on next page)

(continued from previous page)

```
...                                 parallel_config=parallel_config,
...                                 training_config=training_config)
>>> expert = SequentialMLP(num_local_experts=2, config=config)
>>> shape = (32, 64)
>>> tokens_per_expert = [20, 12]
>>> permuted_local_hidden_states = ms.Tensor(np.random.standard_normal(shape).astype(np.
↪float32))
>>> output_local, output_bias_local = expert(permuted_local_hidden_states, tokens_per_expert)
>>> print(output_local.shape)
(32, 64)
```

1.1.21 mindspeed_ms.legacy.model.moe.moe_layer.MoELayer

class mindspeed_ms.legacy.model.moe.moe_layer.**MoELayer**(*config: TransformerConfig*, *submodules=None*, *layer_number: int = None*)

Expert layer.

Parameters

- **config** (`TransformerConfig`) –Configuration object for the transformer model. For details, refer to TransformerConfig class.
- **submodules** (*MLPSubmodules, optional*) –The parameter is reserved and is not currently in use. Default: `None`.
- **layer_number** (*int, optional*) –The parameter is reserved and is not currently in use. Default: `None`.

Inputs:

- **hidden_states** (Tensor) - The input hidden states of the local experts.

Outputs:

Tuple of 2 Tensor.

- **output** (Tensor) - The output of the local experts.
- **mlp_bias** (Tensor) - Not used now.

Raises

- **ValueError** –If *ep_world_size* is less than or equal to `0`.
- **ValueError** –If *num_experts* is not divisible by *ep_world_size*.
- **ValueError** –If the elements of *local_expert_indices* is larger than or equal to `num_experts`.
- **ValueError** –If *moe_config.moe_token_dispatcher_type* is not `alltoall`.
- **ValueError** –If *self.training* is `True` and *get_tensor_model_parallel_world_size()* is larger than `1`, and *self.sp* is not `True`.

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore.communication import init
>>> from mindspeed_ms.core.config import TransformerConfig, ModelParallelConfig,␣
↪TrainingConfig, MoEConfig
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.legacy.model.moe.moe_layer import MoELayer
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE, deterministic='ON')
>>> init()
>>> initialize_model_parallel()
>>> parallel_config = ModelParallelConfig()
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> moe_config = MoEConfig(num_experts=4,
...                        moe_router_topk=2)
>>> config = TransformerConfig(vocab_size=128,
...                            num_layers=1,
...                            num_attention_heads=1,
...                            num_query_groups=1,
...                            hidden_size=64,
...                            ffn_hidden_size=128,
...                            parallel_config=parallel_config,
...                            training_config=training_config,
...                            moe_config=moe_config)
>>> mlp = MoELayer(config)
>>> shape = (8, 2, 64)
>>> hidden_states = ms.Tensor(np.random.standard_normal(shape).astype(np.float32))
>>> output, mlp_bias= mlp(hidden_states)
>>> print(output.shape)
(8, 2, 64)
```

1.1.22 mindspeed_ms.legacy.model.moe.router.TopKRouter

class `mindspeed_ms.legacy.model.moe.router.`**`TopKRouter`**(*config: TransformerConfig*)

TopK router. Calculates scores based on input data and selects top-K experts to process input data.

Parameters

config (`TransformerConfig`) –Configuration object for the transformer model. For details, refer to TransformerConfig class.

Inputs:

- **input** (Tensor) - Input tensor.

Outputs:

Tuple of 2 Tensor.

- **scores** (Tensor) - The probabilities tensor after load balancing.

- **indices** (Tensor) - The indices tensor after top-k selection.

Raises

- `NotImplementedError` –If *moe_config.moe_router_load_balancing_type* is equal to `sinkhorn`.
- `ValueError` –If *moe_config.moe_router_load_balancing_type* is not equal to *sinkhorn*, *aux_loss* or *none*.

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> from mindspeed_ms.core.config import ModelParallelConfig, MoEConfig, TrainingConfig,␣
↪TransformerConfig
>>> from mindspeed_ms.legacy.model.moe.router import TopKRouter
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> import mindspore as ms
>>> import mindspore.common.dtype as mstype
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE, deterministic='ON')
>>> init()
>>> initialize_model_parallel(tensor_model_parallel_size=1,expert_model_parallel_size=1)
>>> parallel_config = ModelParallelConfig(tensor_model_parallel_size=1)
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> moe_cfg = MoEConfig(num_experts=4, moe_router_topk=2)
>>> model_cfg = TransformerConfig(
...     vocab_size=1,
...     num_layers=1,
...     num_attention_heads=1,
...     seq_length=8,
...     hidden_size=16,
...     ffn_hidden_size=64,
...     gated_linear_unit=True,
...     param_init_type=mstype.float32,
...     parallel_config=parallel_config,
...     moe_config=moe_cfg,
...     training_config=training_config,
... )
>>> router = TopKRouter(model_cfg)
>>> data = ms.Tensor(np.random.random((32, 16)).astype(np.float32))
>>> scores, indices = router(data)
>>> print(scores.shape)
(32 ,2)
```

1.1.23 mindspeed_ms.legacy.model.moe.token_dispatcher.MoEAlltoAllTokenDispatcher

class mindspeed_ms.legacy.model.moe.token_dispatcher.**MoEAlltoAllTokenDispatcher**(*num_local_experts: int*, *local_expert_indices: List[int]*, *config: TransformerConfig*)

In the MoE architecture, the MoEAlltoAllTokenDispatcher scheduler is responsible for assigning tokens to various experts for processing, and reassembling the processed results back to the original token order.

Parameters

- **num_local_experts** (*int*) –How many local experts on this rank.
- **local_expert_indices** (*List[int]*) –Indices of local experts on this rank.
- **config** (TransformerConfig) –Configuration object for the transformer model.

Raises

- **ValueError** –If *num_local_experts* is not larger than `0`.
- **ValueError** –If the length of *local_expert_indices* is not equal to `num_local_experts`.

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore.communication import init
>>> from mindspeed_ms.core.config import TransformerConfig, ModelParallelConfig,␣
↪TrainingConfig, MoEConfig
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.legacy.model.moe.token_dispatcher import MoEAlltoAllTokenDispatcher
>>> num_local_experts = 4
>>> ms.set_seed(1024)
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE, deterministic='ON')
>>> init()
>>> initialize_model_parallel()
>>> parallel_config = ModelParallelConfig()
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> moe_config = MoEConfig(num_experts=num_local_experts,
...                        moe_router_topk=2)
>>> config = TransformerConfig(vocab_size=128,
...                            num_layers=1,
...                            num_attention_heads=1,
...                            num_query_groups=1,
...                            hidden_size=64,
...                            ffn_hidden_size=128,
```

(continues on next page)

(continued from previous page)

```
...                            parallel_config=parallel_config,
...                            training_config=training_config,
...                            moe_config=moe_config)
>>> dispatcher = MoEAlltoAllTokenDispatcher(num_local_experts=num_local_experts,
...                                         local_expert_indices=range(num_local_experts),
...                                         config=config)
>>> hidden_states = ms.Tensor(np.random.standard_normal((8, 2, 64)).astype(np.float32))
>>> hidden_states = hidden_states.reshape(-1, hidden_states.shape[-1])
>>> scores_first_column = np.random.rand(16, 1)
>>> complementary_scores = 1 - scores_first_column
>>> scores = ms.Tensor(np.hstack((scores_first_column, complementary_scores)))
>>> indices_array = np.array([np.random.choice(num_local_experts, size=2, replace=False) for␣
↪_ in range(16)])
>>> indices = ms.Tensor(indices_array, dtype=ms.int32)
>>> dispatched_input, tokens_per_expert = dispatcher.token_permutation(hidden_states, scores,
↪ indices)
>>> print(dispatched_input.shape)
(32, 64)
>>> print(tokens_per_expert)
[7 9 7 9]
>>> expert_output = ms.Tensor(np.random.standard_normal((32, 64)).astype(np.float32))
>>> output, _ = dispatcher.token_unpermutation(expert_output, bias=None)
>>> print(output.shape)
(16, 64)
```

1.1.24 mindspeed_ms.legacy.model.ParallelAttention

class mindspeed_ms.legacy.model.**ParallelAttention**(*config*, *layer_number*, *attention_type=AttnType.self_attn*, *attn_mask_type=AttnMaskType.padding*)

This class represents a parallel attention mechanism. It can handle different attention types and is configurable with various parameters.

Parameters

- **config** (*dict*) –Transformer configuration. For details, please refer to TransformerConfig.
- **layer_number** (*int*) –Number which indicates the index of this transformer layer in the whole transformer block.
- **attention_type** (*int, optional*) –Attention type. Support [AttnType::self_attn = 1, AttnType::cross_attn = 2]. Default: `AttnType.self_attn`.
- **attn_mask_type** (*int, optional*) –Attention mask type. Support [AttnMaskType::padding = 1, AttnMaskType::causal = 2]. Default: `AttnMaskType.padding`.

Inputs:

- **hidden_states** (Tensor) - The shape of hidden_states Tensor is (B, S, H).
- **attention_mask** (Tensor) - The shape of attention_mask Tensor is (B, N, S_q, S_k).
- **encoder_output** (Tensor, optional) - Tensor of encoder output used for cross attention. Default: `None`.
- **inference_params** (Tensor, optional) - Tensor of inference params. Currently not supported. Default: `None`.
- **rotary_pos_emb** (Tensor, optional) - Tensor of rotary position embedding. Default: `None`.

Outputs:

- **output** (Tensor) - The shape of output Tensor is (B, S, H).

- **bias** (Tensor) - The trainable bias parameter.

Raises

- `NotImplementedError` –If use flash attention but *attention_type* is not `AttnType.self_attn`.
- `ValueError` –If *group_query_attention* is `True` but the *num_query_groups* is not divisible by *tp_group_size*.
- `ValueError` –If *attention_type* is neither `AttnType::self_attn` nor `AttnType::cross_attn`.
- `NotImplementedError` –If *attention_type* is 2 and *group_query_attention* in config is `True`.
- `ValueError` –If *hidden_size* is not equal to *kv_hidden_size* and *attention_type* is 2.
- `NotImplementedError` –If *get_context_parallel_world_size()* > 1 and *args.context_parallel_algo* is equal to `ulysses_cp_algo` and not use flash attention.
- `NotImplementedError` –If *inference_params* is not `None`.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> from mindspore import Tensor
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.legacy.model import ParallelAttention
>>> from mindspeed_ms.core.config import (ModelParallelConfig,
...                                        TrainingConfig,
...                                        TransformerConfig)
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> class MyNet(nn.Cell):
...     def __init__(self, config):
...         super(MyNet, self).__init__()
...         self.attention = ParallelAttention(layer_number=1, config=config)
...     def construct(self, x, attention_mask):
...         output, _ = self.attention(x, attention_mask)
...         return output
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE,
...     deterministic='ON')
>>> init()
>>> initialize_model_parallel(tensor_model_parallel_size=2)
>>> parallel_config = ModelParallelConfig(tensor_model_parallel_size=2)
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> config = TransformerConfig(vocab_size=1,
>>>                            num_layers=1,
>>>                            num_attention_heads=8,
>>>                            num_query_groups=4,
```

(continues on next page)

(continued from previous page)

```
>>>                             hidden_size=256,
>>>                             ffn_hidden_size=256,
>>>                             parallel_config=parallel_config,
>>>                             training_config=training_config)
>>> input_shape = (32, 1024, 256)
>>> input = Tensor(np.random.standard_normal(input_shape).astype(np.float32))
>>> mask = np.ones((32, 1024, 1024), dtype=np.uint8)
>>> mask = Tensor(np.expand_dims(mask, axis=1))
>>> out = MyNet(config=config)(input, mask)
>>> print(out.shape)
(32, 1024, 256)
```

1.1.25 mindspeed_ms.legacy.model.ParallelLMLogits

class mindspeed_ms.legacy.model.**ParallelLMLogits**(*config*, *bias=False*, *compute_dtype=None*)

Head to get the logits of each token in the vocab.

Parameters

- **config** (*dict*) –Transformer configuration. For details, please refer to TransformerConfig.
- **bias** (*bool, optional*) –Specifies whether the layer uses a bias vector. Default: `False`.
- **compute_dtype** (*dtype.Number, optional*) –The computation type. Default: `None`.

Inputs:

- **input_** (Tensor) - Tensor of hidden states.
- **word_embedding_table** (Parameter) - Weight matrix passed from embedding layer.
- **parallel_output** (bool, optional) - Specifies whether return paralleled output on each tensor parallel rank. Default: `True`.
- **bias** (Tensor, optional) - The trainable bias parameter. Default: `None`.

Outputs:

- **logits_parallel** (Tensor) - If `parallel_output` is `True`, the output is a paralleled logits tensor on each tensor parallel rank, else the output will be a logits tensor gathering all the parallel output.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
```

(continues on next page)

(continued from previous page)

```
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.core.config import ModelParallelConfig, TrainingConfig,␣
↪TransformerConfig
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.legacy.model import ParallelLMLogits
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE, deterministic='ON')
>>> init()
>>> initialize_model_parallel(tensor_model_parallel_size=2)
>>> parallel_config = ModelParallelConfig(tensor_model_parallel_size=2)
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> config = TransformerConfig(seq_length=16,
...                            vocab_size=1,
...                            num_layers=1,
...                            num_attention_heads=8,
...                            num_query_groups=4,
...                            hidden_size=256,
...                            ffn_hidden_size=256,
...                            parallel_config=parallel_config,
...                            training_config=training_config)
>>> model = ParallelLMLogits(config=config, bias=False, compute_dtype=ms.float32)
>>> input = Tensor(np.random.random((2, 3, 6)).astype(np.float32))
>>> word_emb = Tensor(np.random.random((6, 6)).astype(np.float32))
>>> logits = model(input, word_emb, parallel_output=True)
>>> print(logits.shape)
(2, 3, 6)
```

1.1.26 mindspeed_ms.legacy.model.ParallelMLP

class `mindspeed_ms.legacy.model.`**`ParallelMLP`**(*config*, *is_expert=False*)

Implementation of parallel feedforward block.

Parameters

- **`config`** (`TransformerConfig`) –Configuration object for the transformer model.
- **`is_expert`** (*`bool`*, *`optional`*) –This block is an expert block. Default: `False`.

Inputs:

- **hidden_states** (Tensor) - Tensor of shape (B, S, H) or (S, B, H).

Outputs:

- **output** (Tensor) - Output tensor of shape (B, S, H) or (S, B, H).

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.core.config import (ModelParallelConfig,
...                                       TrainingConfig,
...                                       TransformerConfig)
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.legacy.model.transformer import ParallelMLP
>>> ms.set_seed(2024)
>>> init()
>>> initialize_model_parallel()
>>> parallel_config = ModelParallelConfig(tensor_model_parallel_size=1)
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> seq_length=4
>>> dataset_size=1
>>> hidden_size=8
>>> model_config = TransformerConfig(vocab_size=4000,
...                                  num_layers=1,
...                                  num_attention_heads=4,
...                                  hidden_size=hidden_size,
...                                  ffn_hidden_size=hidden_size * 4,
...                                  parallel_config=parallel_config,
...                                  training_config=training_config)
>>> mlp = ParallelMLP(config=model_config)
>>> shape = (seq_length, dataset_size, hidden_size)
>>> input_data = Tensor(np.random.random(shape).astype(np.float32))
>>> output, _ = mlp(input_data)
>>> print(output.shape)
(4, 1, 8)
```

1.1.27 mindspeed_ms.legacy.model.ParallelTransformer

class `mindspeed_ms.legacy.model.`**`ParallelTransformer`**(*config*, *model_type*, *layer_type=LayerType.encoder*, *self_attn_mask_type=AttnMaskType.padding*, *post_norm=True*, *pre_process=False*, *post_process=False*, *drop_path_rate=0.0*)

A transformer block. It consists of multiple transformer layers and can handle various configurations and processing steps.

Parameters

- **config** (*dict*) –Configuration dictionary for the parallel transformer.
- **model_type** (*int*) –Type of the model. Support [ModelType::encoder_or_decoder = 1, ModelType::encoder_and_decoder = 2, ModelType::retro_encoder = 3, ModelType::retro_decoder = 4].
- **layer_type** (*int, optional*) –Type of the layer. Support [LayerType::encoder = 1, LayerType::decoder = 2, LayerType::retro_encoder = 3, LayerType::retro_decoder = 4, LayerType::retro_decoder_with_retriever = 5]. Default: `LayerType.encoder`.

- **`self_attn_mask_type`** (*`int`*, *`optional`*) –Attention mask type. Support [AttnMaskType::padding = 1, AttnMaskType::causal = 2]. Default: `AttnMaskType.padding`.
- **`post_norm`** (*`bool`*, *`optional`*) –Insert normalization layer at the end of transformer block. Default: `True`.
- **`pre_process`** (*`bool`*, *`optional`*) –When using pipeline parallel, indicate whether it's the first stage. Default: `False`.
- **`post_process`** (*`bool`*, *`optional`*) –When using pipeline parallel, indicate whether it's the last stage. Default: `False`.
- **`drop_path_rate`** (*`float`*, *`optional`*) –Drop path rate. Currently not supported if greater than 0. Default: `0.0`.

Inputs:

- **hidden_states** (Tensor) - The shape of hidden_states tensor is (B, S, H).
- **attention_mask** (Tensor) - Tensor of attention mask.
- **encoder_output** (Tensor, optional) - Encoder output tensor. Currently not supported. Default: `None`.
- **enc_dec_attn_mask** (Tensor, optional) - Encoder-decoder attention mask tensor. Currently not supported. Default: `None`.
- **retriever_input** (Tensor, optional) - Retriever input tensor. Currently not supported. Default: `None`.
- **retriever_output** (Tensor, optional) - Retriever output tensor. Currently not supported. Default: `None`.
- **retriever_attn_mask** (Tensor, optional) - Retriever attention mask tensor. Currently not supported. Default: `None`.
- **inference_params** (Tensor, optional) - Tensor of inference params. Currently not supported. Default: `None`.
- **rotary_pos_emb** (Tensor, optional) - Tensor of rotary position embedding. Default: `None`.

Outputs:

- **hidden_states** (Tensor) - The shape of hidden_states tensor is (B, S, H).

Raises

- **`NotImplementedError`** –If *drop_path_rate* greater than `0`.
- **`NotImplementedError`** –If *distribute_saved_activations* in config is true and *sequence_parallel* in config is `False`.
- **`NotImplementedError`** –If *transformer_impl* in config is `transformer_engine`.
- **`NotImplementedError`** –If *fp8* in config is not `None`.
- **`NotImplementedError`** –If *retro_add_retriever* in config is `True`.
- **`NotImplementedError`** –If *model_type* equal to `3` or `4`.
- **`NotImplementedError`** –If *encoder_output*, *enc_dec_attn_mask*, *retriever_input*, *retriever_output*, *retriever_attn_mask* or *inference_params* is not none.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import mindspore.common.dtype as mstype
>>> from mindspore import Tensor
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.legacy.model import ParallelTransformer
>>> from mindspeed_ms.core.config import (ModelParallelConfig,
...                                        TrainingConfig,
...                                        TransformerConfig)
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> class MyNet(nn.Cell):
...     def __init__(self, config):
...         super(MyNet, self).__init__()
...         self.transformer = ParallelTransformer(config=config, model_type=None)
...     def construct(self, x, attention_mask):
...         output = self.transformer(x, attention_mask)
...         return output
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE, deterministic='ON')
>>> init()
>>> initialize_model_parallel(tensor_model_parallel_size=2)
>>> parallel_config = ModelParallelConfig(tensor_model_parallel_size=2)
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> config = TransformerConfig(seq_length=16,
>>>                            vocab_size=1,
>>>                            num_layers=1,
>>>                            num_attention_heads=8,
>>>                            num_query_groups=4,
>>>                            hidden_size=256,
>>>                            ffn_hidden_size=256,
>>>                            parallel_config=parallel_config,
>>>                            training_config=training_config)
>>> input_shape = (32, 1024, 256)
>>> input = Tensor(np.random.standard_normal(input_shape).astype(np.float32))
>>> mask = Tensor(np.triu(np.ones((1024, 1024)), 1), mstype.uint8)
>>> out = MyNet(config=config)(input, mask).shape
>>> print(out)
(32, 1024, 256)
```

1.1.28 mindspeed_ms.legacy.model.ParallelTransformerLayer

class `mindspeed_ms.legacy.model.`**`ParallelTransformerLayer`**(*config*, *layer_number*, *layer_type=LayerType.encoder*, *self_attn_mask_type=AttnMaskType.padding*, *drop_path_rate=0.0*)

A single transformer layer. It combines normalization, attention, cross attention, and an MLP to process input hidden states.

Parameters

- **config** (*dict*) –Configuration dictionary for the transformer layer.
- **layer_number** (*int*) –Number which indicates the index of this transformer layer in the whole transformer block.
- **layer_type** (*int, optional*) –Type of the layer. Support [LayerType::encoder = 1, LayerType::decoder = 2, LayerType::retro_encoder = 3, LayerType::retro_decoder = 4, LayerType::retro_decoder_with_retriever = 5]. Default: `LayerType.encoder`.
- **self_attn_mask_type** (*int, optional*) –Attention mask type. Support [AttnMaskType::padding = 1, AttnMaskType::causal = 2]. Default: `AttnMaskType.padding`.
- **drop_path_rate** (*float, optional*) –Drop path rate. Currently not supported if greater than 0. Default: `0.0`.

Inputs:

- **hidden_states** (Tensor) - The shape of hidden_states tensor is (B, S, H).
- **attention_mask** (Tensor) - Tensor of attention mask.
- **encoder_output** (Tensor, optional) - Encoder output tensor. Currently not supported. Default: `None`.
- **enc_dec_attn_mask** (Tensor, optional) - Encoder-decoder attention mask tensor. Currently not supported. Default: `None`.
- **retriever_input** (Tensor, optional) - Retriever input tensor. Currently not supported. Default: `None`.
- **retriever_output** (Tensor, optional) - Retriever output tensor. Currently not supported. Default: `None`.
- **retriever_attn_mask** (Tensor, optional) - Retriever attention mask tensor. Currently not supported. Default: `None`.
- **inference_params** (Tensor, optional) - Tensor of inference params. Currently not supported. Default: `None`.
- **rotary_pos_emb** (Tensor, optional) - Tensor of rotary position embedding. Default: `None`.

Outputs:

- **output** (Tensor) - The shape of output tensor is (B, S, H).

Raises

- `NotImplementedError` –If 'bias_dropout_fusion' in config is true.
- `NotImplementedError` –If *drop_path_rate* greater than 0.
- `NotImplementedError` –If 'retro_add_retriever' in config is true.
- `NotImplementedError` –If *encoder_output*, *enc_dec_attn_mask*, *retriever_input*, *retriever_output*, *retriever_attn_mask* or *inference_params* is not none.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import mindspore.common.dtype as mstype
>>> from mindspore import Tensor
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.legacy.model import ParallelTransformerLayer
>>> from mindspeed_ms.core.config import ModelParallelConfig, TrainingConfig,␣
↪TransformerConfig
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> class MyNet(nn.Cell):
...     def __init__(self, config):
...         super(MyNet, self).__init__()
...         self.layer = ParallelTransformerLayer(layer_number=1, config=config)
...     def construct(self, x, attention_mask):
...         output = self.layer(x, attention_mask)
...         return output
...
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE, deterministic='ON')
>>> init()
>>> initialize_model_parallel(tensor_model_parallel_size=2)
>>> parallel_config = ModelParallelConfig(tensor_model_parallel_size=2)
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> config = TransformerConfig(vocab_size=1,
>>>                            num_layers=1,
>>>                            num_attention_heads=8,
>>>                            num_query_groups=4,
>>>                            hidden_size=256,
>>>                            ffn_hidden_size=256,
>>>                            parallel_config=parallel_config,
>>>                            training_config=training_config)
>>> input_shape = (32, 1024, 256)
>>> input = Tensor(np.random.standard_normal(input_shape).astype(np.float32))
>>> mask = Tensor(np.triu(np.ones((1024, 1024)), 1), mstype.uint8)
>>> out = MyNet(config=config)(input, mask).shape
>>> print(out)
(32, 1024, 256)
```

1.1.29 mindspeed_ms.legacy.model.RotaryEmbedding

class mindspeed_ms.legacy.model.**RotaryEmbedding**(*kv_channels*, *rotary_percent=1.0*, *rotary_interleaved=False*, *seq_len_interpolation_factor=None*, *rotary_base=10000*)

Rotary positional embedding for language model.

Parameters

- **kv_channels** (*int*) –Projection weights dimension in multi-head attention. Obtained from transformer config.
- **rotary_percent** (*float*, *optional*) –Percent of rotary dimension to use for rotary position embeddings. Default: `1.0`.
- **rotary_interleaved** (*bool*, *optional*) –Determines the method of applying rotary embeddings to the input dimensions. Default: `False`.
- **seq_len_interpolation_factor** (*float*, *optional*) –scale of linearly interpolating RoPE for longer sequences. The value must be a float larger than 1.0. Default: `None`.
- **rotary_base** (*int*, *optional*) –Base period for rotary position embeddings. Default: `10000`.

Inputs:

- **max_seq_len** (int) - Max sequence length of inputs.
- **offset** (int) - The starting point for the position encoding.

Outputs:

- **emb** (Tensor) - Embeddings after applying RoPE.

Raises

NotImplementedError –If *rotary_interleaved* is `True`.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore.nn as nn
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.communication import init
>>> from mindspeed_ms.legacy.model.rotary_pos_embedding import RotaryEmbedding
>>> from mindspeed_ms.core.config import ModelParallelConfig, TrainingConfig,␣
↪TransformerConfig
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> class MyNet(nn.Cell):
...     def __init__(self, config: TransformerConfig):
...         super(MyNet, self).__init__()
```

(continues on next page)

(continued from previous page)

```
...         self.rotary_embedding = RotaryEmbedding(config.seq_length)
...     def construct(self, x: Tensor):
...         emb = self.rotary_embedding(x.shape[1])
...         return emb
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE, deterministic='ON')
>>> init()
>>> initialize_model_parallel(tensor_model_parallel_size=1, context_parallel_size=1)
>>> parallel_config = ModelParallelConfig(tensor_model_parallel_size=1)
>>> training_config = TrainingConfig(parallel_config=parallel_config)
>>> config = TransformerConfig(seq_length=16,
...                            vocab_size=1,
...                            num_layers=1,
...                            num_attention_heads=16,
...                            hidden_size=256,
...                            ffn_hidden_size=256,
...                            parallel_config=parallel_config,
...                            training_config=training_config)
>>> bs = 2
>>> seq_len = 16
>>> hidden_size = 256
>>> input_shape = (bs, seq_len, hidden_size)
>>> net = MyNet(config)
>>> input = Tensor(np.random.standard_normal(input_shape).astype(np.float32))
>>> output = net(input)
>>> print(output.shape)
(16, 1, 1, 16)
```

1.1.30 mindspeed_ms.legacy.model.transformer.CoreAttention

class `mindspeed_ms.legacy.model.transformer.`**`CoreAttention`**(*layer_number*, *config*, *attn_mask_type=AttnMaskType.padding*)

Core attention mechanism for calculating attention weights and context representations between queries, key and value layers.

Parameters

- **`layer_number`** (*int*) –Number which indicates the index of this transformer layer in the whole transformer block.
- **`config`** (*dict*) –A configuration dictionary that provides various settings for the attention mechanism.
- **`attn_mask_type`** (*int, optional*) –Attention mask type. Support [AttnMaskType::padding = 1, AttnMaskType::causal = 2]. Default: `1`.

Inputs:

- **query_layer** (Tensor) - Query layer. The shape of query Tensor is (S, B, N, D).
- **key_layer** (Tensor) - Key layer. The shape of key Tensor is (S, B, N, D).
- **value_layer** (Tensor) - Value layer. The shape of value Tensor is (S, B, N, D).
- **attention_mask** (Tensor) - Attention mask. The shape of attention mask Tensor is (B, S_q, S_k).

Outputs:

- **context_layer** (Tensor) - The shape of context_layer Tensor is (S, B, H).

Raises

`NotImplementedError` –If *masked_softmax_fusion* in config is True.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> from mindspore import Tensor
>>> from mindspore.communication.management import init
>>> from mindspeed_ms.legacy.model.transformer import CoreAttention
>>> from mindspeed_ms.core.config import ModelParallelConfig, TransformerConfig
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> class MyNet(nn.Cell):
...     def __init__(self, config):
...         super(MyNet, self).__init__()
...         self.core_attn = CoreAttention(layer_number=1, config=config)
...     def construct(self, x, mask):
...         out = self.core_attn(x, x, x, mask)
...         return out
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE, deterministic='ON')
>>> init()
>>> initialize_model_parallel(tensor_model_parallel_size=2)
>>> parallel_config = ModelParallelConfig(tensor_model_parallel_size=2)
>>> config = TransformerConfig(vocab_size=1,
...                            num_layers=1,
...                            num_attention_heads=8,
...                            num_query_groups=4,
...                            hidden_size=256,
...                            ffn_hidden_size=256,
...                            parallel_config=parallel_config,
...                            training_config=None)
>>> input_shape = (1024, 1, 4, 32)
>>> input = Tensor(np.random.standard_normal(input_shape).astype(np.float32))
>>> mask = np.ones((1, 1024, 1024), dtype=np.uint8)
>>> mask = Tensor(np.expand_dims(mask, axis=1))
>>> out = MyNet(config=config)(input, mask)
>>> print(out.shape)
(1024, 1, 128)
```

1.1.31 mindspeed_ms.legacy.model.TransformerLanguageModel

class mindspeed_ms.legacy.model.**TransformerLanguageModel**(*config*, *encoder_attn_mask_type*, *num_tokentypes=0*, *add_encoder=True*, *add_decoder=False*, *decoder_attn_mask_type=AttnMaskType.causal*, *add_pooler=False*, *pre_process=True*, *post_process=True*, *visual_encoder=None*, ***kwargs*)

Transformer language model.

Parameters

- **config** (TransformerConfig) –The transformer configuration includes init_method, parallel_config, etc.
- **encoder_attn_mask_type** (*int*) –Encoder attention mask type.
- **num_tokentypes** (*int, optional*) –If > 0, using tokentypes embedding. Default: `0`.
- **add_encoder** (*bool, optional*) –If True, use encoder. Default: `True`.
- **add_decoder** (*bool, optional*) –If True, use decoder. Default: `False`.
- **decoder_attn_mask_type** (*int, optional*) –Decoder attention mask type. Default: `AttnMaskType.causal`.
- **add_pooler** (*bool, optional*) –If True, use pooler. Default: `False`.
- **pre_process** (*bool, optional*) –When using pipeline parallel, indicate whether it's the first stage. Default: `True`.
- **post_process** (*bool, optional*) –When using pipeline parallel, indicate whether it's the last stage. Default: `True`.
- **visual_encoder** (*nn.Cell, optional*) –Visual encoder. Default: `None`.
- **kwargs** (*dict*) –Other input.

Inputs:

- **enc_input_ids** (Tensor) - Encoder input indexes. Shape (B, S).
- **enc_position_ids** (Tensor) - Encoder position offset. Shape (B, S).
- **enc_attn_mask** (Tensor) - Encoder attention mask. Shape (B, S).
- **dec_input_ids** (Tensor, optional) - Decoder input indexes. Shape (B, S). Default: `None`.
- **dec_position_ids** (Tensor, optional) - Decoder input position indices. Shape (B, S). Default: `None`.
- **dec_attn_mask** (Tensor, optional) - Decoder attention mask. Shape (B, S). Default: `None`.
- **retriever_input_ids** (Tensor, optional) - Retriever input token indices. Shape: Depends on the input shape of the retrieval task. Default: `None`.
- **retriever_position_ids** (Tensor, optional) - Retriever input position indices. Shape: Depends on the input shape of the retrieval task. Default: `None`.
- **retriever_attn_mask** (Tensor, optional) - Retriever attention mask. Used to control the attention range in the retriever when calculating attention. Shape: Depends on the attention calculation shape of the retriever. Default: `None`.
- **enc_dec_attn_mask** (Tensor, optional) - Encoder-decoder attention mask. Shape: Depends on the attention calculation between the encoder and decoder. Default: `None`.
- **tokentype_ids** (Tensor, optional) - List of token type ids to be fed to a model. Shape (B, S). Default: `None`.

- **inference_params** (InferenceParams, optional) - Inference parameters. Used to specify specific settings during inference, such as maximum generation length, max batch size, etc. Default: `None`.
- **pooling_sequence_index** (int, optional) - Pooling sequence index. Default: `0`.
- **enc_hidden_states** (Tensor, optional) - Encoder hidden states. Shape: Depends on the output shape of the encoder. Default: `None`.
- **output_enc_hidden** (bool, optional) - Whether to output encoder hidden states. Default: `False`.
- **input_image** (Tensor, optional) - Tensor of the input image. Shape $(N, C_{in}, H_{in}, W_{in})$ or $(N, H_{in}, W_{in}, C_{in},)$ depending on *data_format*. Default: `None`.
- **delimiter_position** (Tensor, optional) - Delimiter position tensor. Shape (B, N), where N represents the number of delimiters. Default: `None`.
- **image_embedding** (Tensor, optional) - Image embedding tensor. The shape depends on the dimension of the image embedding, for example (batch_size, embedding_dim). Default: `None`.

Outputs:

- **encoder_output** - Output Tensor of shape (B, S, H) or (S, B, H).

Raises

- **`ValueError`** –If config.untie_embeddings_and_output_weights and add_decoder are both True.
- **`RuntimeError`** –If the length of the input is not 1.
- **`NotImplementedError`** –If *config.retro_add_retriever* is True.
- **`NotImplementedError`** –if *visual_encoder* or *add_decoder* is True.
- **`NotImplementedError`** –If *dec_input_ids*, *dec_position_ids*, *dec_attn_mask*, *retriever_input_ids*, *retriever_position_ids*, *retriever_attn_mask*, *enc_dec_attn_mask*, *input_image*, *delimiter_position* or *image_embedding* is not None.
- **`NotImplementedError`** –if *output_enc_hidden* is True.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import os
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.communication import init
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.legacy.model import TransformerLanguageModel
>>> from mindspeed_ms.core.config import (
...     init_configs_from_yaml,
```

(continues on next page)

(continued from previous page)

```
...     TrainingConfig,
...     ModelParallelConfig,
...     TransformerConfig,
...     DatasetConfig,
... )
>>> os.environ['HCCL_BUFFSIZE'] = "1"
>>> CONFIG_PATH = "test_language_model.yaml"
>>> training_config, parallel_config, dataset_config, model_config = init_configs_from_yaml(
...     CONFIG_PATH, [TrainingConfig, ModelParallelConfig, DatasetConfig, TransformerConfig]
... )
>>> ms.set_context(device_target="Ascend", mode=ms.PYNATIVE_MODE, deterministic="ON")
>>> init()
>>> initialize_model_parallel()
>>> batch_size = dataset_config.batch_size
>>> sq = model_config.seq_length
>>> language_model = TransformerLanguageModel(model_config, encoder_attn_mask_type=None)
>>> input_data = Tensor(np.random.random((batch_size, sq)).astype(np.int32))
>>> attention_mask = Tensor(np.zeros((batch_size, 1, sq, sq)).astype(np.int32))
>>> hidden_states = language_model(input_data, None, attention_mask)
>>> print(hidden_states.shape)
(8, 32, 64)
```

1.1.32 mindspeed_ms.training.loss_func.LossWithMask

class `mindspeed_ms.training.loss_func.`**`LossWithMask`**(*loss_func*, **args*, ***kwargs*)

Calculate the loss with mask and mean reduction.

Parameters

- **loss_func** (*Function*) –Loss function.
- **args** (*tuple*) –Input arguments.
- **kwargs** (*dict*) –Extra keyword configuration arguments.

Inputs:

- **logits** (Tensor) - The output logits of the backbone. Tensor of shape (N, C). Data type must be float16 or float32.
- **label** (Tensor) - The ground truth label of the sample. Tensor of shape $(N,)$ or the same shape as *logits*.
- **input_mask** (Tensor) - The *input_mask* indicates whether there are padded inputs and for padded inputs it will not be counted into loss. Tensor of shape $(N,)$.

Outputs:

- The corresponding cross entropy loss.

Examples

Note: Before running the following examples, you need to configure the environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore.communication.management import init
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, nn
>>> from mindspeed_ms.core.parallel_state import initialize_model_parallel
>>> from mindspeed_ms.training.loss_func import LossWithMask
>>> ms.set_context(device_target='Ascend', mode=ms.PYNATIVE_MODE)
>>> ms.set_seed(2024)
>>> init()
>>> initialize_model_parallel()
>>> loss = LossWithMask(nn.CrossEntropyLoss())
>>> logits = Tensor(np.array([[3, 5, 6, 9, 12, 33, 42, 12, 32, 72]]),
...                 mstype.float32)
>>> labels = Tensor(np.array([1]).astype(np.int32))
>>> input_mask = Tensor(np.ones(1).astype(np.float32))
>>> output = loss(logits, labels, input_mask)
>>> print(output)
67.0
```

1.1.33 mindspeed_ms.training.training.pretrain

`mindspeed_ms.training.training.`**`pretrain`**(*train_valid_test_datasets_provider*, *model_provider_func*, *model_type*, *forward_step_func=None*, *process_non_loss_data_func=None*, ***kwargs*)

Pre-training interface. You can use this interface to transfer data iterators, model definitions, and configuration items to start model training.

Parameters

- **`train_valid_test_datasets_provider`** (*`function`*) –A function that takes the size of train/valid/test dataset and returns *train, valid, test* datasets. Currently not in use.
- **`model_provider_func`** (*`function`*) –A function that returns the model.
- **`model_type`** (*`enum`*) –An enum that specifies the type of model being trained. Currently not in use.
- **`forward_step_func`** (*`function, optional`*) –Forward step function. Currently not in use. Default: `None`.
- **`process_non_loss_data_func`** (*`function, optional`*) –A function to post process outputs of the network. Currently not in use. Default: `None`.
- **`kwargs`** (*`dict`*) –Other input. The data iterator and configuration *all_config* are transferred from here.

Raises

`ValueError` –If *all_config* is not passed in with kwargs.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspeed_ms.training.training import pretrain
>>> from mindspeed_ms.core.config import TransformerConfig
>>> def model_provider_func(pre_process=True, post_process=True):
...     network_with_loss = GPTModel(
...         all_config.model_config,
...         pre_process=pre_process,
...         post_process=post_process
...     )
...     return network_with_loss
>>> all_config = TransformerConfig()
>>> pretrain(
...     train_valid_test_datasets_provider=None,
...     model_provider_func=model_provider_func,
...     model_type=None,
...     all_config=all_config
... )
```