ModelCenter

BMTrain Team

Apr 05, 2022

GETTING STARTED

ModelCenter implements PLMs (Pretrained Language Models) based on BMTrain backend.

CHAPTER

ONE

MAIN ADVANTAGES:

- Low-Resource
- Efficient
- Extendable

1.1 Installation

1.1.1 1. From PyPI (Recommend)

```
$ pip install model-center
```

1.1.2 2. From Source

```
$ git clone https://github.com/OpenBMB/ModelCenter.git
$ cd ModelCenter
$ pip install -r requirements.txt
$ python3 setup.py install
```

1.2 Quick start

In the quick start, you will walk through how to fine-tune a BERT model on a classification task.

1.2.1 Initialize bmtrain backend

First, you need to import `bmtrain` and use `bmtrain.init_distributed()` at the beginning of your code, which can initialize the distributed environments.

```
import bmtrain as bmt
bmt.init_distributed(seed=0)
```

1.2.2 Prepare the model

Next, you can simply get a pre-trained BERT model from `model_center`, e.g., *bert-base-uncased*. When fine-tuning BERT on the classification task, a feed-forward layer need to be appended to the last layer.

```
import torch
from model_center.model import Bert, BertConfig
from model_center.layer import Linear

class BertModel(torch.nn.Module):
    def __init__(self, config):
        super().__init__()
        self.bert = Bert.from_pretrained("bert-base-uncased")
        self.dense = Linear(config.dim_model, 2)
        bmt.init_parameters(self.dense)

    def forward(self, input_ids, attention_mask):
        pooler_output = self.bert(input_ids=input_ids, attention_mask=attention_mask).
↪pooler_output
        logits = self.dense(pooler_output)
        return logits

config = BertConfig.from_pretrained("bert-base-uncased")
model = BertModel(config)
```

1.2.3 Perpare the dataset

The next step is to prepare the dataset used for training and evaluation. Here, we use the BoolQ dataset from the SuperGLUE benchmark. You need to download the dataset and put the unzipped folder in `your_path_to_dataset`.

```
from model_center.dataset.bertdataset import DATASET
from model_center.dataset import DistributedDataLoader
from model_center.tokenizer import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
splits = ['train', 'dev']
dataset = {}

for split in splits:
    dataset[split] = DATASET['BoolQ']('your_path_to_dataset', split, bmt.rank(), bmt.
↪world_size(), tokenizer, max_encoder_length=512)

batch_size = 64
train_dataloader = DistributedDataLoader(dataset['train'], batch_size=batch_size,␣
↪shuffle=True)
dev_dataloader = DistributedDataLoader(dataset['dev'], batch_size=batch_size,␣
↪shuffle=False)
```

1.2.4 Train the model

Now, select optimizer, learning rate scheduler, loss function, and then start training the model! Here, we train BERT for 5 epochs and evaluate it at the end of each epoch.

```
optimizer = bmt.optim.AdamOffloadOptimizer(model.parameters())

lr_scheduler = bmt.lr_scheduler.Noam(
    optimizer,
    start_lr = 1e-5,
    warmup_iter = 100,
    end_iter = -1)

loss_func = bmt.loss.FusedCrossEntropy(ignore_index=-100)

for epoch in range(5):
    model.train()
    for data in train_dataloader:
        input_ids = data['input_ids']
        attention_mask = data['attention_mask']
        labels = data['labels']

        optimizer.zero_grad()

        # model forward
        logits = model(input_ids, attention_mask)

        # calculate loss
        loss = loss_func(logits.view(-1, logits.shape[-1]), labels.view(-1))

        # scale loss to avoid precision underflow of fp16
        loss = optimizer.loss_scale(loss)

        # model backward
        loss.backward()

        # clip gradient norm
        grad_norm = bmt.optim.clip_grad_norm(optimizer.param_groups, max_norm=10.0,␣
↪scale = optimizer.scale, norm_type = 2)

        bmt.optim_step(optimizer, lr_scheduler)

        # print information only on rank 0 when distributed training
        # use bmt.sum_loss(loss) to gather all loss information from all distributed␣
↪processes
        bmt.print_rank(
            "loss: {:.4f} | lr: {:.4e}, scale: {:10.4f} | grad_norm: {:.4f} |".format(
                bmt.sum_loss(loss).item(),
                lr_scheduler.current_lr,
                int(optimizer.scale),
                grad_norm,
            )
        )
```

(continues on next page)

(continued from previous page)

```
    # evaluate model
    model.eval()
    with torch.no_grad():
        pd = [] # prediction
        gt = [] # ground_truth
        for data in dev_dataloader:
            input_ids = data["input_ids"]
            attention_mask = data["attention_mask"]
            labels = data["labels"]

            logits = model(input_ids, attention_mask)
            loss = loss_func(logits.view(-1, logits.shape[-1]), labels.view(-1))

            logits = logits.argmax(dim=-1)

            pd.extend(logits.cpu().tolist())
            gt.extend(labels.cpu().tolist())

        # gather results from all distributed processes
        pd = bmt.gather_result(torch.tensor(pd).int()).cpu().tolist()
        gt = bmt.gather_result(torch.tensor(gt).int()).cpu().tolist()

        # calculate metric
        from sklearn.metrics import accuracy_score
        acc = accuracy_score(gt, pd)
        bmt.print_rank(f"accuracy: {acc*100:.2f}")
```

1.2.5 Run your code

You can run the above code using the same launch command as the distributed module of PyTorch.

Choose one of the following commands depending on your version of PyTorch.

- `${MASTER_ADDR}` means the IP address of the master node.
- `${MASTER_PORT}` means the port of the master node.
- `${NNODES}` means the total number of nodes.
- `${GPU_PER_NODE}` means the number of GPUs per node.
- `${NODE_RANK}` means the rank of this node.

torch.distributed.launch

```
$ python3 -m torch.distributed.launch --master_addr ${MASTER_ADDR} --master_port $
↪{MASTER_PORT} --nproc_per_node ${GPU_PER_NODE} --nnodes ${NNODES} --node_rank ${NODE_
↪RANK} train.py
```

torchrun

```
$ torchrun --nnodes=${NNODES} --nproc_per_node=${GPU_PER_NODE} --rdzv_id=1 --rdzv_
↪backend=c10d --rdzv_endpoint=${MASTER_ADDR}:${MASTER_PORT} train.py
```

For more information, please refer to the documentation.

1.3 Benchmark

1.3.1 Comparison between Hugging Face Transformers

Make Big Models trainable on consumer GPUs

Tested on 32GB V100 machine using `bert-large-uncased`, we have comparable throughput as Hugging Face Transformers but much fewer GPU memory footprint.

Tested on a **single consumer GPU**, 11GB 2080Ti, however, training `bert-large-uncased` is no longer supported in Hugging Face Transformers, but we make it possible.

Make Huge Models train easily.

Tested on 40GB A100 machine using `T5-11B`, we make it possible to train with 16 batch-size using two GPUs.

1.3.2 Comparison between Deepspeed ZeRO

see also BMTrain's Performance

1.4 How to write a new model

1.4.1 Model Implementation

We implement our models in `model_center/model`

We provided commonly used modules in `model_center/layer`, such as `Linear`, `LayerNorm`, `Embedding`, which are implemented based on bmtrain.DistributedParameter and bmtrain.DistributedModule, for distributed training support.

We have also implemented common ways of combining modules in `model_center/layer`, which are block. For example, `SelfAttentionBlock` combines Layernorm, Attention, Add&Norm together. Each blocks has diverse option, e.g., `FFNBlock` supports `gated_relu`, `relu`, `gated_gelu`, `gelu`; blocks support pre-layernorm and post-layernorm.

With the help of these commonly used modules we provided, a new model can be written easily without many exceptions. You can just add the model specific feature into the common structure.

A classic transformer is implemented in the following structure:

We use bmtrain.CheckpointBlock, and bmtrain.TransformerBlockList to wrap our transformer blocks. These reducd the GPU memory usage by a great amount without adding lots of computation time. For more information, see BMTrain's Quick Start

```
T5(
  (input_embedding): Embedding()
  (position_bias_enc): RelativePositionEmbedding()
  (position_bias_dec): RelativePositionEmbedding()
  (encoder): Encoder(
    (layers): bmtrain.TransformerBlockList(
      (0): bmtrain.CheckpointBlock(
        TransformerBlock(
          (self_att): SelfAttentionBlock(
            (layernorm_before_attention): LayerNorm()
            (attention): Attention(
              (project_q): Linear()
              (project_k): Linear()
              (project_v): Linear()
              (attention_out): Linear()
            )
          )
          (ffn): FFNBlock(
            (layernorm_before_ffn): LayerNorm()
            (ffn): FeedForward(
              (w_in): DenseACT(
                (w): Linear()
                (act): ReLU()
              )
              (w_out): Linear()
            )
          )
        )
      )
      (1): bmtrain.CheckpointBlock()
      .
      .
      .
    )
    (output_layernorm): LayerNorm()
  )
  (decoder): Decoder(
    (layers): bmtrain.TransformerBlockList(
      (0): bmtrain.CheckpointBlock(
        (self_att): SelfAttentionBlock(
          (layernorm_before_attention): LayerNorm()
          (attention): Attention(
            (project_q): Linear()
            (project_k): Linear()
            (project_v): Linear()
            (attention_out): Linear()
          )
        )
```

(continues on next page)

(continued from previous page)

```
        (cross_att): CrossAttentionBlock(
          (layernorm_before_attention): LayerNorm()
          (attention): Attention(
            (project_q): Linear()
            (project_k): Linear()
            (project_v): Linear()
            (attention_out): Linear()
          )
        )
        (ffn): FFNBlock(
          (layernorm_before_ffn): LayerNorm()
          (ffn): FeedForward(
            (w_in): DenseACT(
              (w): Linear()
              (act): ReLU()
            )
            (w_out): Linear()
          )
        )
      )
      (1): bmtrain.CheckpointBlock()
      .
      .
      .
    )
    (output_layernorm): LayerNorm()
  )
  (output_projection): Linear(
    (weight): bmtrain.DistributedParameter()
    (bias): bmtrain.DistributedParameter()
  )
)
```

1.4.2 Model Config

We add model configs in `model_center/model/config`

By inheriting `model_center.config.Config`, config class can parse json files with `config.from_json_file(path)` method, the parsed json file are then save to the config class and used by model by instantiating model with `model(config)`.

1.5 Pretrain data processing

1.5.1 1. Raw data source

First, prepare raw data into `data_1.txt`, `data_2.txt`, ...

In the raw text files, use line breaks to separate different datas, i.e., each line has one data (for example, a long document). Notice that there may be line breaks inside data, they should be replaced with unique identifier, such as <n>, and replaced back in step 2.

Examples:

```
The Lord of the Rings<n>The Lord of the Rings is an epic high-fantasy novel by English␣
↪author and scholar J. R. R. Tolkien. Set in Middle-earth, intended to be Earth at some␣
↪distant time in the past, the story began as a sequel to Tolkien's 1937 children's␣
↪book The Hobbit, but eventually developed into a much larger work. Written in stages␣
↪between 1937 and 1949, The Lord of the Rings is one of the best-selling books ever␣
↪written, with over 150 million copies sold.<n>The title refers to the story's main␣
↪antagonist, the Dark Lord Sauron, who in an earlier age created the One Ring to rule␣
↪the other Rings of Power given to Men, Dwarves, and Elves, in his campaign to conquer␣
↪all of Middle-earth. From homely beginnings in the Shire, a hobbit land reminiscent of␣
↪the English countryside, the story ranges across Middle-earth, following the quest to␣
↪destroy the One Ring mainly through the eyes of the hobbits Frodo, Sam, Merry and␣
↪Pippin.
The Little Prince<n>I thought that I was rich, with a flower that was unique in all the␣
↪world; and all I had was a common rose. A common rose...<n>To me, you are still␣
↪nothing more than a little boy who is just like a hundred thousand other little boys.␣
↪And I have no need of you. And you, on your part, have no need of me. To you, I am␣
↪nothing more than a fox like a hundred thousand other foxes. But if you tame me, then␣
↪we shall need each other. To me, you will be unique in all the world. To you, I shall␣
↪be unique in all the world.<n>The wheat fields have nothing to say to me. And that is␣
↪sad. But you have hair that is the color of gold. Think how wonderful that will be␣
↪when you have tamed me! The grain, which is also golden, will bring me back the␣
↪thought of you. And I shall love to listen to the wind in the wheat.
...
```

Important: Since Streaming data processing and training are used in subsequent steps, data should be pre-shuffled before putting into `data.txt` sequentially.

1.5.2 2. Tokenize

- Implement an Encoder, which gives it a line of input data and it returns you the tokenized result.
- Mapping all datas with Encoder, with the help of multiprocessing
- We also provide a tools called `indexed_dataset`, which compress the tokenized data into binary format.

There is an example code in `model_center/tools/preprocess_cpm1_lm.py`, we simplify it and made it into a example like the following:

```python
import multiprocessing
from model_center.tools import indexed_dataset

# 1. Implement an Encoder, which gives it a line of input data and it returns you the␣
↪tokenized result.
class Encoder(object):
    def initializer(self):
        Encoder.tokenizer = YourTokenizer()

    def encode(self, line):
        data = line.strip().replace("<n>", "\n") # replace back line break symbol

        doc_ids = Encoder.tokenizer.encode(data)
```

(continues on next page)

(continued from previous page)

```
        max_length = 512 # model's maximum input length

        pieces = []
        while i < len(doc_ids): # split document into chunks with model's maximum length
            piece = doc_ids[i:i+max_length]
            if len(piece) < 32: # drop too short chunks
                break
            i += max_length

            pieces.append(piece)

        return pieces

if __name__ == '__main__':
    # assumes that there are 100 raw data files, named `data_1.txt` to `data_100.txt`
    for ith in range(1, 101):
        fin = open(f"data_{ith}.txt", "r", encoding="utf-8")

        # encoder use the tokenizer to encode data
        encoder = Encoder()

        # 2. Mapping all datas with Encoder, with the help of multiprocessing
        pool = multiprocessing.Pool(processes=64, initializer=encoder.initializer)
        encoded_docs = pool.imap_unordered(encoder.encode, fin, chunksize=10)

        # 3. tool `indexed_dataset` compress the tokenized data into binary format `bin_
↪file`
        # it will also generate another small `idx_file` for saving meta information in␣
↪order to decode `bin_file`.
        bin_file = os.path.join("path/to/your/tokenized_output_folder/", f"tokenized_
↪{ith}.bin")
        idx_file = os.path.join("path/to/your/tokenized_output_folder/", f"tokenized_
↪{ith}.idx")

        binary_builder = indexed_dataset.make_builder(bin_file, impl="mmap", dtype=np.
↪uint16)

        # put tokenized data into binary_builder
        for pieces in encoded_docs:
            for doc_ids in pieces:
                binary_builder.add_item(torch.IntTensor(doc_ids))

        # finish compressing tokenized data into `bin_file`, and generate meta␣
↪information into `idx_file`
        binary_builder.finalize(idx_file)

        # close multiproceessing mapping
        pool.close()
```

1.5.3 3. Self-Supervised Dataset

We provide a tool `model_center.dataset.DistributedMMapIndexedDataset` for loading those compressed binary files and do extra processing, for example, a commonly used way is to randomly mask a span and ask the model to regenerate it.

It is important to note that, tokenization is an rather slow operation, it may become a bottleneck if tokenization is perform while the model is being trained. Thus, we put tokenization into data pre-processing stage (step 2), which is run beforehand.

However, We do not calculate things like `attention_mask` and save them into binary files because it will lead to multiplication of space occupation, which is not affordable for the large amount of data needed for training large language models. Putting those extra information that are less time consuming into `__getitem__` function of pytorch Dataset, space will be saved since the space will be recycled when the next batch arrives.

```
from model_center.dataset import DistributedMMapIndexedDataset

class YourDataset(torch.utils.data.Dataset):
    def __init__(self, ctx : MMapIndexedDataset):
        self.ctx = ctx

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

    def __get_item_data(self, input_ids): # do extra processing
        length = input_ids.shape[0]

        # randomly mask a span in range [lef, rig)
        lef = random.randint(input_length)
        rig = random.randint(lef, input_length)

        # pretrain objective: regenerate the masked span and ignore other positions
        targets = np.full((input_length), -100) # -100 as ignore_index
        targets[lef:rig] = input_ids[lef:rig]

        # calculate attention_mask, to tell model which positions are visible
        attention_mask = (np.arange((input_length)) < lef) | (np.arange((input_length)) >
↪= rig)
        return input_ids, targets, input_length, attention_mask

    def __getitem__(self, ith):
        input_ids = self.ctx[ith] # get the i-th data from DistributedMMapIndexedDataset
        return self.__get_item_data(ctx) # do extra processing and return

if __name__ == '__main__':
    dataset = YourDataset(
        DistributedMMapIndexedDataset("path/to/your/tokenized_output_folder", "tokenized
↪", bmt.rank(), bmt.world_size()),
        # the second argument "tokenized" is the common prefix of your tokenized file␣
↪name,
        # here we assumes that they are called "tokenized_1.bin", "tokenized_2.idx", etc.
    )
```

1.6 BERT

Bert

We currently support loading the following checkpoint via `Bert.from_pretrained(identifier)`

- bert-base-cased
- bert-base-uncased
- bert-large-cased
- bert-large-uncased
- bert-base-chinese
- bert-base-multilingual-cased

1.6.1 BertConfig

class model_center.model.**BertConfig**(*vocab_size=119547*, *type_size=2*, *position_size=512*, *dim_model=768*, *num_heads=12*, *dim_head=64*, *dim_ff=3072*, *num_layers=12*, *dropout_p=0.1*, *emb_init_mean=0.0*, *emb_init_std=1*, *pos_bias_type='none'*, *position_bias_max_distance=1024*, *norm_init_var=1.0*, *norm_bias=True*, *norm_eps=1e-12*, *att_init_mean=0.0*, *att_init_std=0.02*, *att_bias=True*, *att_mask_value=- 10000.0*, *ffn_init_mean=0.0*, *ffn_init_std=0.02*, *ffn_bias=True*, *ffn_activate_fn='gelu'*, *proj_init_mean=0.0*, *proj_init_std=1*, *proj_bias=True*, *length_scale=False*, *attn_scale=True*, *half=True*, *int8=False*, *tied=False*, *cls_head=None*, *post_layer_norm=True*)

This is a configuration class that stores the configuration of the BERT model, which inherits from the Config class. It is used to instantiate the Bert model according to the specified parameters and define the model architecture. You can set specific parameters to control the output of the model.

For example: [*dim_model*] is used to determine the Dimension of the encoder layers and the pooler layer. You can choose to use the default value of 768 or customize their dimensions.

1.6.2 BertModel

class model_center.model.**Bert**(*config:* model_center.model.config.bert_config.BertConfig)

forward(*input_ids=None*, *length=None*, *attention_mask=None*, *token_type_ids=None*, *position_ids=None*, *head_mask=None*, *inputs_embeds=None*, *encoder_hidden_states=None*, *encoder_attention_mask=None*, *output_attentions=None*, *output_hidden_states=None*, *return_dict=True*, *return_logits=False*)

This model inherits from BaseModel. This model is also a PyTorch torch.nn.Module subclass. You can use it as a regular PyTorch Module. You can also select the data and data type that you want the model to return through changing the value of *return_dict* and *return_logits*.

Parameters

- **input_ids** (`torch.Tensor` of shape `(batch, seq_length)`) – Indices of input sequence tokens. It will be embedded by model's internal embedding lookup matrix.

- **length** (torch.Tensor of shape (batch)) – Length of input sequence before padding.
- **attention_mask** (torch.Tensor of shape (batch, seq_length)) – Used to avoid performing attention on padding token indices.
- **token_type_ids** (torch.Tensor of shape (batch, seq_length)) – Unused.
- **position_ids** (torch.Tensor of shape (batch, seq_length)) – Unused.
- **head_mask** (torch.Tensor of shape (num_layers, num_heads)) – Unused.
- **inputs_embeds** (torch.Tensor of shape (batch, seq_length, dim_model)) – Embedding of the input. You can choose to directly pass the inputs embedding to control the way of embedding.
- **encoder_hidden_states** (torch.Tensor of shape(batch, seq_length, dim_model)) – Unused.
- **encoder_attention_mask** (torch.Tensor of shape (batch, seq_length)) – Unused.
- **output_attentions** (torch.Tensor of shape (batch, num_heads, seq_length, seq_length)) – Unused.
- **output_hidden_states** (torch.Tensor of shape (batch, seq_length, dim_model)) – Unused.
- **return_dict** (bool) – Whether to return a BaseModelOutputWithPoolingAndCrossAttentions instead of just a tuple.
- **return_logits** (bool) – Whether to return the prediction score for each token in vocabulary (before softmax).

Returns The Bert output. Depended on the value of *return_dict* and *return_logits*

Return type BaseModelOutputWithPoolingAndCrossAttentions or tuple or torch.Tensor of shape (batch, seq_length, vocab_output_size) or (batch, seqlen, cls_head)

1.6.3 BertTokenizer

class model_center.tokenizer.**BertTokenizer**

The current implementation is mainly an alias to BertTokenizer of Hugging Face Transformers. we will change to our SAM implementation in the future, which will be a more efficient tokenizer.

1.7 GPT2

GPT2

We currently support loading the following checkpoint via GPT2.from_pretrained(identifier)

- gpt2-base
- gpt2-medium
- gpt2-large
- gpt2-xl

1.7.1 GPT2Config

class model_center.model.**GPT2Config**(*vocab_size=50258*, *dim_model=768*, *num_heads=12*, *dim_head=64*, *dim_ff=3072*, *num_layers=12*, *dropout_p=0.1*, *emb_init_mean=0.0*, *emb_init_std=1*, *pos_bias_type='none'*, *position_size=1024*, *norm_init_var=1.0*, *norm_bias=True*, *norm_eps=1e-05*, *att_init_mean=0.0*, *att_init_std=0.02*, *att_bias=True*, *att_mask_value=- 10000.0*, *ffn_init_mean=0.0*, *ffn_init_std=0.02*, *ffn_bias=True*, *ffn_activate_fn='gelu'*, *proj_init_mean=0.0*, *proj_init_std=1*, *proj_bias=True*, *length_scale=False*, *attn_scale=True*, *half=True*, *int8=False*, *tied=True*, *cls_head=None*, *post_layer_norm=False*)

This is a configuration class that stores the configuration of the GPT-2 model, which inherits from the Config class. It is used to instantiate the Bert model according to the specified parameters and define the model architecture. You can set specific parameters to control the output of the model.

For example: [*dim_model*] is used to determine the Dimension of the encoder layers. You can choose to use the default value of 768 or customize their dimensions.

1.7.2 GPT2Model

class model_center.model.**GPT2**(*config:* model_center.model.config.gpt2_config.GPT2Config)

forward(*input_ids=None*, *length=None*, *attention_mask=None*, *token_type_ids=None*, *position_ids=None*, *head_mask=None*, *inputs_embeds=None*, *encoder_hidden_states=None*, *encoder_attention_mask=None*, *output_attentions=None*, *output_hidden_states=None*, *return_dict=True*, *return_logits=False*)

The GPT-2 Model transformer outputs raw hidden-states or logits as you want. This model inherits from BaseModel. This model is also a PyTorch torch.nn.Module subclass.You can use it as a regular PyTorch Module. You can also select the data and data type that you want the model to return through changing the value of *return_dict* and *return_logits*.

Parameters

- **input_ids** (`torch.Tensor` of shape `(batch, seq_length)`) – Indices of input sequence tokens. It will be embedded by model's internal embedding lookup matrix.
- **length** (`torch.Tensor` of shape `(batch)`) – Length of input sequence before padding.
- **attention_mask** (`torch.Tensor` of shape `(batch, seq_length)`) – Used to avoid performing attention on padding token indices.
- **token_type_ids** (`torch.Tensor` of shape `(batch, seq_length)`) – Unused.
- **position_ids** (`torch.Tensor` of shape `(batch, seq_length)`) – Unused.
- **head_mask** (`torch.Tensor` of shape `(num_layers, num_heads)`) – Unused.
- **inputs_embeds** (`torch.Tensor` of shape `(batch, seq_length, dim_model)`) – Embedding of the input. You can choose to directly pass the inputs embedding to control the way of embedding.
- **encoder_hidden_states** (`torch.Tensor` of shape(batch, seq_length, dim_model)) – Unused.
- **encoder_attention_mask** (`torch.Tensor` of shape(batch, seq_length)) – Unused.

- **output_attentions** (torch.Tensor of shape (batch, num_heads, seq_length, seq_length)) – Unused.
- **output_hidden_states** (torch.Tensor of shape (batch, seq_dec, dim_model)) – Unused.
- **return_dict** (bool) – Whether to return a BaseModelOutputWithPastAndCrossAttentions instead of just a tuple.
- **return_logits** (bool) – Whether to return the prediction score for each token in vocabulary (before softmax).

Returns The GPT-2 output. Depended on the value of *return_dict* and *return_logits*

Return type BaseModelOutputWithPastAndCrossAttentions or tuple or torch.Tensor of shape (batch, seq_dec, vocab_output_size) or (batch, seqlen, cls_head)

1.7.3 GPT2Tokenizer

class model_center.tokenizer.**GPT2Tokenizer**

The current implementation is mainly an alias to GPT2Tokenizer of Hugging Face Transformers. we will change to our SAM implementation in the future, which will be a more efficient tokenizer.

1.8 GPT-j

GPTj

We currently support loading the following checkpoint via `GPTj.from_pretrained(identifier)`

- gptj-6b

1.8.1 GPTjConfig

class model_center.model.**GPTjConfig**(*vocab_size=50400*, *dim_model=4096*, *num_heads=16*, *dim_head=256*, *dim_ff=16384*, *num_layers=28*, *dropout_p=0*, *emb_init_mean=0.0*, *emb_init_std=1*, *pos_bias_type='rotary'*, *pos_rotary_dim=64*, *norm_init_var=1.0*, *norm_bias=True*, *norm_eps=1e-05*, *att_init_mean=0.0*, *att_init_std=0.1*, *att_bias=False*, *att_mask_value=- inf*, *ffn_init_mean=0.0*, *ffn_init_std=0.1*, *ffn_bias=True*, *ffn_activate_fn='gelu'*, *proj_init_mean=0.0*, *proj_init_std=1*, *proj_bias=True*, *length_scale=False*, *attn_scale=True*, *half=True*, *int8=False*, *tied=False*, *cls_head=None*, *post_layer_norm=False*)

This is a configuration class that stores the configuration of the GPT-J model, which inherits from the Config class. It is used to instantiate the Bert model according to the specified parameters and define the model architecture. You can set specific parameters to control the output of the model.

For example: [*dim_model*] is used to determine the Dimension of the encoder layers. You can choose to use the default value of 4096 or customize their dimensions.

1.8.2 GPTjModel

class model_center.model.**GPTj**(*config:* model_center.model.config.gptj_config.GPTjConfig)

forward(*input_ids=None*, *length=None*, *attention_mask=None*, *token_type_ids=None*, *position_ids=None*, *head_mask=None*, *inputs_embeds=None*, *output_attentions=None*, *output_hidden_states=None*, *return_dict=True*, *return_logits=False*)

The GPT-J Model transformer outputs raw hidden-states or logits as you want. This model inherits from BaseModel. This model is also a PyTorch torch.nn.Module subclass.You can use it as a regular PyTorch Module. You can also select the data and data type that you want the model to return through changing the value of *return_dict* and *return_logits*.

Parameters

- **input_ids** (torch.Tensor of shape (batch, seq_length)) – Indices of input sequence tokens. It will be embedded by model's internal embedding lookup matrix.
- **length** (torch.Tensor of shape (batch)) – Length of input sequence before padding.
- **attention_mask** (torch.Tensor of shape (batch, seq_length)) – Used to avoid performing attention on padding token indices.
- **token_type_ids** (torch.Tensor of shape (batch, seq_length)) – Unused.
- **position_ids** (torch.Tensor of shape (batch, seq_length)) – Unused.
- **head_mask** (torch.Tensor of shape (num_layers, num_heads)) – Unused.
- **inputs_embeds** (torch.Tensor of shape (batch, seq_length, dim_model)) – Embedding of the input. You can choose to directly pass the inputs embedding to control the way of embedding.
- **output_attentions** (torch.Tensor of shape (batch, num_heads, seq_length, seq_length)) – Unused.
- **output_hidden_states** (torch.Tensor of shape (batch, seq_dec, dim_model)) – Unused.
- **return_dict** (bool) – Whether to return a BaseModelOutputWithPastAndCrossAttentions instead of just a tuple.
- **return_logits** (bool) – Whether to return the prediction score for each token in vocabulary (before softmax).

Returns The GPT-J output. Depended on the value of *return_dict* and *return_logits*

Return type BaseModelOutputWithPastAndCrossAttentions or tuple or torch.Tensor of shape (batch, seq_dec, vocab_output_size) or (batch, seqlen, cls_head)

1.8.3 GPTjTokenizer

class model_center.tokenizer.**GPTjTokenizer**

The current implementation is mainly an alias to AutoTokenizer of Hugging Face Transformers. we will change to our SAM implementation in the future, which will be a more efficient tokenizer.

1.9 T5

T5

We currently support loading the following checkpoint via `T5.from_pretrained(identifier)`

- t5-small
- t5-base
- t5-large
- t5-3b
- t5-11b

1.9.1 T5Config

class model_center.model.**T5Config**(*vocab_size=32128*, *dim_model=768*, *num_heads=12*, *dim_head=64*, *dim_ff=3072*, *num_encoder_layers=12*, *num_decoder_layers=12*, *dropout_p=0*, *emb_init_mean=0.0*, *emb_init_std=1*, *pos_bias_type='relative'*, *position_bias_num_buckets=32*, *position_bias_max_distance=128*, *pos_init_mean=0.0*, *pos_init_std=1*, *norm_init_var=1.0*, *norm_bias=False*, *norm_eps=1e-06*, *att_init_mean=0.0*, *att_init_std=1*, *att_bias=False*, *att_mask_value=-inf*, *ffn_init_mean=0.0*, *ffn_init_std=1*, *ffn_bias=False*, *ffn_activate_fn='relu'*, *proj_init_mean=0.0*, *proj_init_std=1*, *proj_bias=False*, *length_scale=False*, *attn_scale=False*, *half=True*, *int8=False*, *tied=False*, *cls_head=None*, *post_layer_norm=False*)

This is a configuration class that stores the configuration of the model, which inherits from the Config class. It is used to instantiate the Bert model according to the specified parameters and define the model architecture. You can set specific parameters to control the output of the model.

For example: [*dim_model*] is used to determine the Dimension of the encoder layers. You can choose to use the default value of 768 or customize their dimensions.

1.9.2 T5Model

class model_center.model.**T5**(*config:* model_center.model.config.t5_config.T5Config)

forward(*input_ids=None*, *length=None*, *decoder_input_ids=None*, *decoder_length=None*, *attention_mask=None*, *decoder_attention_mask=None*, *head_mask=None*, *decoder_head_mask=None*, *cross_attn_head_mask=None*, *encoder_outputs=None*, *inputs_embeds=None*, *decoder_inputs_embeds=None*, *output_attentions=None*, *output_hidden_states=None*, *return_dict=True*, *return_logits=False*)

T5 is an encoder-decoder model and converts problems into a text-to-text format. This model inherits from BaseModel. This model is also a PyTorch torch.nn.Module subclass. You can use it as a regular PyTorch Module. You can also select the data and data type that you want the model to return through changing the value of *return_dict* and *return_logits*.

Parameters

- **input_ids** (`torch.Tensor` of shape `(batch, seq_enc)`) – Indices of input sequence tokens. It will be embedded by model's internal embedding lookup matrix.

- **length** (torch.Tensor of shape (batch)) – Length of input sequence before padding.
- **attention_mask** (torch.Tensor of shape (batch, seq_enc)) – Used to avoid performing attention on padding token indices in input.
- **decoder_input_ids** (torch.Tensor of shape (batch, seq_enc)) – Indices of decoder input sequence tokens .
- **decoder_length** (torch.Tensor of shape (batch)) – Length of decoder input sequence before padding.
- **deocoder_attention_mask** (torch.Tensor of shape (batch, seq_enc)) – Used to avoid performing attention on padding token indices in decoder input.
- **head_mask** (torch.Tensor of shape (num_layers, num_heads)) – Unused.
- **decoder_head_mask** (torch.Tensor of shape (num_layers, num_heads)) – Unused.
- **cross_attn_head_mask** (torch.Tensor of shape (num_layers, num_heads)) – Unused.
- **encoder_outputs** (torch.Tensor of shape (batch, dim_model, seq_enc)) – Outputs of encoder.
- **inputs_embeds** (torch.Tensor of shape (batch, seq_enc, dim_model)) – Embedding of the input. You can choose to directly pass the inputs embedding to control the way of embedding.
- **decoder_inputs_embeds** (torch.Tensor of shape (batch, seq_dec, dim_model)) – Embedding of the decoder input. You can choose to directly pass the inputs embedding to control the way of embedding.
- **output_attentions** (torch.Tensor of shape (batch, num_heads, seq_enc, seq_enc)) – Unused.
- **output_hidden_states** (torch.Tensor of shape (batch, seq_dec, dim_model)) – Unused.
- **return_dict** (bool) – Whether to return a Seq2SeqModelOutput instead of just a tuple.
- **return_logits** (bool) – Whether to return the prediction score for each token in vocabulary (before softmax).

Returns The T5 output. Depended on the value of *return_dict* and *return_logits*

Return type Seq2SeqModelOutput or tuple or torch.Tensor of shape (batch, seq_dec, vocab_output_size) or (batch, seqlen, cls_head)

1.9.3 T5Tokenizer

class model_center.tokenizer.**T5Tokenizer**

The current implementation is mainly an alias to T5Tokenizer of Hugging Face Transformers. we will change to our SAM implementation in the future, which will be a more efficient tokenizer.

1.10 CPM1

CPM1

1.10.1 CPM1Config

class model_center.model.**CPM1Config**(*vocab_size=30968*, *dim_model=768*, *num_heads=12*, *dim_head=64*, *dim_ff=256*, *num_layers=12*, *dropout_p=0*, *emb_init_mean=0.0*, *emb_init_std=1*, *pos_bias_type='relative'*, *position_bias_num_buckets=32*, *position_bias_max_distance=128*, *pos_init_mean=0.0*, *pos_init_std=1*, *norm_init_var=1.0*, *norm_bias=False*, *norm_eps=1e-06*, *att_init_mean=0.0*, *att_init_std=0.02*, *att_bias=False*, *att_mask_value=- inf*, *ffn_init_mean=0.0*, *ffn_init_std=0.02*, *ffn_bias=False*, *ffn_activate_fn='gated_gelu'*, *proj_init_mean=0.0*, *proj_init_std=1*, *proj_bias=False*, *length_scale=False*, *attn_scale=False*, *half=True*, *int8=False*, *tied=False*, *cls_head=None*, *post_layer_norm=False*)

This is a configuration class that stores the configuration of the CPM model, which inherits from the Config class. It is used to instantiate the Bert model according to the specified parameters and define the model architecture. You can set specific parameters to control the output of the model.

For example: [*dim_model*] is used to determine the Dimension of the encoder layers. You can choose to use the default value of 768 or customize their dimensions.

1.10.2 CPM1Model

class model_center.model.**CPM1**(*config:* model_center.model.config.cpm1_config.CPM1Config)

forward(*input: torch.Tensor*, *length: torch.Tensor*, *context: torch.Tensor*, *span: torch.Tensor*)

This model inherits from BaseModel. This model is also a PyTorch torch.nn.Module subclass. You can use it as a regular PyTorch Module.

Parameters

- **input** (torch.Tensor of shape (batch, seqlen)) –
- **length** (torch.Tensor of shape (batch)) –
- **context** (torch.Tensor of shape (batch, seqlen)) –
- **span** (torch.Tensor of shape (batch, seqlen)) –

Returns The CPM output. Prediction scores of the language modeling before SoftMax.

Return type torch.Tensor of shape (batch, seqlen, vocab_size) or (batch, seqlen, cls_head)

1.10.3 CPM1Tokenizer

class model_center.tokenizer.**CPM1Tokenizer**(*vocab_file*, *max_len=None*, *q2b=False*, *eod_token='<eod>'*, *pad_token='<pad>'*, *unk_token='<unk>'*, *line_token='</n>'*, *space_token='</_>'*)

decode(*tokens*)

Decode ids into a string.

encode(*text*)

Encode a string into ids.

tokenize(*text*)

Tokenize a string.

1.11 CPM2

CPM2

1.11.1 CPM2Config

class model_center.model.**CPM2Config**(*vocab_size=26240*, *dim_model=768*, *num_heads=12*, *dim_head=64*, *dim_ff=256*, *num_encoder_layers=12*, *num_decoder_layers=12*, *dropout_p=0*, *emb_init_mean=0.0*, *emb_init_std=1*, *pos_bias_type='relative'*, *position_bias_num_buckets=32*, *position_bias_max_distance=128*, *pos_init_mean=0.0*, *pos_init_std=1*, *norm_init_var=1.0*, *norm_bias=False*, *norm_eps=1e-06*, *att_init_mean=0.0*, *att_init_std=0.02*, *att_bias=False*, *att_mask_value=- inf*, *ffn_init_mean=0.0*, *ffn_init_std=0.02*, *ffn_bias=False*, *ffn_activate_fn='gated_gelu'*, *proj_init_mean=0.0*, *proj_init_std=1*, *proj_bias=False*, *length_scale=False*, *attn_scale=False*, *half=True*, *int8=False*, *cls_head=None*, *post_layer_norm=False*)

This is a configuration class that stores the configuration of the CPM-2 model, which inherits from the Config class. It is used to instantiate the Bert model according to the specified parameters and define the model architecture. You can set specific parameters to control the output of the model.

For example: [*dim_model*] is used to determine the Dimension of the encoder layers. You can choose to use the default value of 768 or customize their dimensions.

1.11.2 CPM2Model

class model_center.model.**CPM2**(*config:* model_center.model.config.cpm2_config.CPM2Config)

forward(*enc_input: torch.Tensor*, *enc_length: torch.Tensor*, *dec_input: torch.Tensor*, *dec_length: torch.Tensor*)

This model inherits from BaseModel. This model is also a PyTorch torch.nn.Module subclass. You can use it as a regular PyTorch Module.

Parameters

- **enc_input** (torch.Tensor of shape (batch, seq_enc)) – Indices of input sequence tokens for encoder. It will be embedded by model's internal embedding lookup matrix.
- **enc_length** (torch.Tensor of shape (batch)) – Length of input sequence for encoder before padding.
- **dec_input** (torch.Tensor of shape (batch, seq_dec)) – Indices of input sequence tokens for decoder. It will be embedded by model's internal embedding lookup matrix.
- **dec_length** (torch.Tensor of shape (batch)) – Length of input sequence for encoder before padding.

Returns The CPM-2 output. Prediction scores of the language modeling before SoftMax.

Return type torch.Tensor of shape (batch, seq_dec, vocab_output_size) or (batch, seqlen, cls_head)

1.11.3 CPM2Tokenizer

class model_center.tokenizer.**CPM2Tokenizer**(*vocab_file*, *max_sentinels=190*, *max_len=None*, *q2b=False*, *sod_token='<s>'*, *eod_token='<eod>'*, *pad_token='<pad>'*, *unk_token='<unk>'*, *line_token='</n>'*, *space_token='</_>'*)

decode(*tokens*)

Decode ids into a string.

encode(*text*)

Encode a string into ids.

tokenize(*text*)

Tokenize a string.

1.12 module

1.12.1 Linear

class model_center.layer.**Linear**(**args: Any*, ***kwargs: Any*)

Bases: bmtrain.DistributedModule

A fully connected layer, which performs $\boldsymbol{y} = \mathbf{W}\boldsymbol{x} + \boldsymbol{b}$

Parameters

- **dim_in** (*int*) – input dimension of $\boldsymbol{x}$
- **dim_out** (*int*) – output dimension of $\boldsymbol{y}$
- **dtype** (*optional*) – Defaults to torch.half.
- **init_mean** (*float, optional*) – mean of $\mathbf{W} \sim \mathcal{N}(\text{mean}, \text{std}^2)$. Defaults to 0.
- **init_std** (*float, optional*) – std of $\mathbf{W} \sim \mathcal{N}(\text{mean}, \text{std}^2)$. Defaults to 1.
- **bias** (*bool, optional*) – whether to add bias term $\boldsymbol{b}$. Defaults to False.

forward(*x: torch.Tensor*)

Parameters **x** (`torch.Tensor` of shape `(batch, seq_len, dim_in)`) – The input of linear layer

Returns The output of the linear transform y.

Return type `torch.Tensor` of shape `(batch, seq_len, dim_out)`

1.12.2 Embedding

class model_center.layer.**Embedding**(**args: Any*, ***kwargs: Any*)

Bases: `bmtrain.DistributedModule`

Embed a sequence of indices through a embedding lookup matrix $\mathbf{W}$.

Parameters

- **vocab_size** (*int*) – indices be in range $[0,$

forward(*ids: torch.Tensor*)

Parameters **ids** (`torch.Tensor` of shape `(batch_size, seq_len)`) – Indices of input sequence tokens.

Returns The embedding output.

Return type `torch.Tensor` of shape `(batch_size, seq_len, embedding_size)`

projection(*x: torch.Tensor*)

Projection based on embedding's weight. For example, embedding map vocab_size to embed_size, than projection map embed_size back to vocab_size.

Parameters **x** (`torch.Tensor` of shape `(batch, seq_len, dim_model)`) – Input of projection

Returns The projection output.

Return type `torch.Tensor` of shape `(batch, seq_len, vocab_output_size)`

1.12.3 RelativePositionEmbedding

class model_center.layer.**RelativePositionEmbedding**(**args: Any*, ***kwargs: Any*)

Bases: `bmtrain.DistributedModule`

Relative Position Embedding

Parameters

- **num_heads** (*int*) – number of heads used in attention module.
- **num_buckets** (*int, optional*) – Defaults to 32.
- **max_distance** (*int, optional*) – Defaults to 128.
- **bidirectional** (*bool, optional*) – Defaults to False.
- **dtype** (*optional*) – Defaults to torch.half.
- **init_mean** (*float, optional*) – Defaults to 0.0.
- **init_std** (*float, optional*) – Defaults to 1.

forward(*query_len*, *key_len*)

Provides relative position embeddings for key and query of *num_heads* attention heads.

Parameters

- **query_len** (int) – Length of query.
- **key_len** (int) – Length of key.

Returns Relative position embedding.

Return type torch.Tensor of shape (num_heads, query_len, key_len)

1.12.4 RotaryEmbedding

class model_center.layer.**RotaryEmbedding**(*rotary_dim: int*)

Bases: torch.nn.modules.module.Module

Rotary Position Embedding

Parameters **rotary_dim** (*int*) – rotary dimension

forward(*h_q*, *h_k*)

Parameters

- **h_q** – (batch_size*num_head, len_q, dim_head)
- **h_k** – (batch_size*num_head, len_k, dim_head)

Returns (batch_size*num_head, len_q, dim_head) h_k : (batch_size*num_head, len_k, dim_head)

Return type h_q

1.12.5 LayerNorm

class model_center.layer.**LayerNorm**(**args: Any*, ***kwargs: Any*)

Bases: bmtrain.DistributedModule

LayerNorm if bias = True: $y = \frac{x-\mathrm{E}[x]}{\mathrm{Var}[x]+\mathrm{eps}} * w + \mathrm{bias}$

RMS LayerNorm if bias = False: $y = \frac{x}{\mathrm{Var}[x]+\mathrm{eps}} * w$

Parameters

- **dim_norm** (*int*) – norm dimesion
- **dtype** (*optional*) – Defaults to torch.half.
- **bias** (*bool, optional*) – whether to add the bias term. Defaults to True.
- **eps** (*float, optional*) – eps term. Defaults to 1e-5.
- **init_var** (*float, optional*) – weight will be all initialized to init_var. Defaults to 1.0.

forward(*x: torch.Tensor*)

Parameters **x** (torch.Tensor of shape (batch_size, seq_len, dim_norm)) – Input tensor that need to be normalized.

Returns The layernorm output.

Return type torch.Tensor of shape (batch_size, seq_len, dim_norm)

1.12.6 Attention

class model_center.layer.**Attention**(**args: Any*, ***kwargs: Any*)

Bases: bmtrain.DistributedModule

attention module consisting procedure of Q, K, V combination and its output projection. For more detail, see Attention is All you Need.

Parameters

- **dim_in** (*int*) – input dimension.
- **dim_head** (*int*) – dimension of each heads used in attention.
- **num_heads** (*int*) – number of heads used in attention.
- **dim_out** (*int, optional*) – output dimension. Defaults to None, which means dim_in = dim_out.
- **dtype** (*optional*) – Defaults to torch.half.
- **init_mean** (*float, optional*) – mean of $\mathbf{W} \sim \mathcal{N}(\text{mean}, \text{std}^2)$ for fully-connected module used in attetion module. Defaults to 0.
- **init_std** (*float, optional*) – std of $\mathbf{W} \sim \mathcal{N}(\text{mean}, \text{std}^2)$ for fully-connected module used in attention module. Defaults to 0.02.
- **bias** (*bool, optional*) – whether to use bias term in fully-connected layers used in attention module. Defaults to False.
- **mask_value** (*float, optional*) – mask value of the masked position. Defaults to *-inf*.
- **pos_bias_type** (*str, optional*) – *relative* for relative position bias, *rotary* for ratery position embedding. Defaults to *none*.
- **attn_scale** (*bool, optional*) – whether to scale before softmax, i.e., softmax($\frac{QK^T}{\sqrt{}}$

forward(*query: torch.Tensor*, *key_value: torch.Tensor*, *mask: torch.Tensor*, *position_bias: Optional[torch.Tensor] = None*)

This model inherits from bmt.DistributedModule.

Parameters

- **query** (torch.Tensor of shape (batch, len_q, dim_model)) – Indices of input sequence tokens. It will be embedded by model's internal embedding lookup matrix.
- **key_value** (torch.Tensor of shape (batch, len_k, dim_model)) – Length of input sequence before padding.
- **mask** (torch.Tensor of shape (batch, len_q, len_k)) – Used to avoid performing attention on padding token indices.
- **position_bias** (torch.Tensor of shape (num_heads, len_q, len_k) or (1, num_heads, len_k, len_q)) – Provide positional information about tensor *key_value* and *query*.

Returns The attention output.

Return type out (torch.Tensor of shape (batch, len_q, dim_model))

1.12.7 FeedForward

class model_center.layer.**FeedForward**(**args: Any*, ***kwargs: Any*)

Bases: bmtrain.DistributedModule

FeedForward module

Parameters

- **dim_in** (*int*) – input dimension.
- **dim_ff** (*int*) – middle dimension.
- **dim_out** (*int, optional*) – output dimension. Defaults to None, which means dim_in = dim_out.
- **dtype** (*optional*) – Defaults to torch.half.
- **init_mean** (*float, optional*) – mean of $\mathbf{W} \sim \mathcal{N}(\text{mean}, \text{std}^2)$ for fully-connected module used in feed-forward layer. Defaults to 0.
- **init_std** (*float, optional*) – std of $\mathbf{W} \sim \mathcal{N}(\text{mean}, \text{std}^2)$ for fully-connected module used in feed-forward layer. Defaults to 0.02.
- **bias** (*bool, optional*) – whether to use bias term in fully-connected layers used in feed-forward module. Defaults to False.
- **activate_fn** (*str, optional*) – Defaults to *gated_gelu*.
- **dropout_p** (*int, optional*) – Defaults to 0.

forward(*x: torch.Tensor*)

Parameters **x** (torch.Tensor of shape (batch, seq_len, dim_in)) – The input of feed-forward module.

Returns The output of feed-forward module.

Return type torch.Tensor of shape (batch, seq_len, dim_out)

1.13 block

1.13.1 Encoder

class model_center.layer.**Encoder**(*num_layers: int*, *dim_model: int*, *dim_ff: int*, *num_heads: int*, *dim_head: int*, *dtype: torch.dtype = torch.float16*, *int8: bool = False*, *norm_init_var: float = 1.0*, *norm_bias: bool = False*, *norm_eps: float = 1e-05*, *att_init_mean: float = 0.0*, *att_init_std: float = 0.02*, *att_bias: bool = False*, *att_mask_value: float = - inf*, *ffn_init_mean: float = 0.0*, *ffn_init_std: float = 0.02*, *ffn_bias: bool = False*, *ffn_activate_fn: str = 'gated_gelu'*, *pos_bias_type: str = 'none'*, *post_layer_norm: bool = False*, *length_scale: bool = False*, *attn_scale: bool = False*, *dropout_p: float = 0*, *parallel_ffn: bool = False*)

Bases: torch.nn.modules.module.Module

Layers of encoder transformer blocks plus an final layernorm.

Parameters

- **num_layers** (*int*) – number of layers.

- **dim_model** (*int*) – main dimension of modules in transformer blocks.
- **dim_ff** (*int*) – dim_ff used in *model_center.layer.FeedForward*.
- **num_heads** (*int*) – num_heads used in *model_center.layer.Attention*.
- **dim_head** (*int*) – dim_head used in *model_center.layer.Attention*.
- **dtype** (*optional*) – Defaults to torch.half.
- **norm_init_var** (*float, optional*) – init_var used in *model_center.layer.LayerNorm*. Defaults to 1.0.
- **norm_bias** (*bool, optional*) – bias used in *model_center.layer.LayerNorm*. Defaults to False.
- **norm_eps** (*float, optional*) – eps used in *model_center.layer.LayerNorm*. Defaults to 1e-5.
- **att_init_mean** (*float, optional*) – init_mean used in *model_center.layer.Attention*. Defaults to 0.0.
- **att_init_std** (*float, optional*) – init_std used in *model_center.layer.Attention*. Defaults to 0.02.
- **att_bias** (*bool, optional*) – bias used in in *model_center.layer.Attention*. Defaults to False.
- **att_mask_value** (*float, optional*) – mask_value used in in *model_center.layer.Attention*. Defaults to float("-inf").
- **ffn_init_mean** (*float, optional*) – init_mean used in *model_center.layer.FeedForward*. Defaults to 0.0.
- **ffn_init_std** (*float, optional*) – init_std used in *model_center.layer.FeedForward*. Defaults to 0.02.
- **ffn_bias** (*bool, optional*) – bias used in *model_center.layer.FeedForward*. Defaults to False.
- **ffn_activate_fn** (*str, optional*) – activate_fn used in *model_center.layer.FeedForward*. Defaults to "gated_gelu".
- **pos_bias_type** (*str, optional*) – pos_bias_type used in *model_center.layer.Attention*. Defaults to "none".
- **post_layer_norm** (*bool, optional*) – whether to use post-layernorm. Defaults to False, which means pre-layernorm.
- **attn_scale** (*bool, optional*) – attn_scale used in in *model_center.layer.Attention*. Defaults to False.
- **dropout_p** (*float, optional*) – Defaults to 0.

forward(*hidden_states: torch.Tensor*, *attention_mask: torch.Tensor*, *position_bias: Optional[torch.Tensor] = None*)

Parameters

- **hidden-states** (`torch.Tensor` of shape `(batch, seq_enc, dim_model)`) – Input of encoder, might be the embedding of a batch of sequences.
- **attention_mask** (`torch.Tensor` of shape `(batch, seq_enc, seq_enc)`) – Avoid invalid areas to participate in the calculation
- **position_bias** (`torch.Tensor` of shape `(num_heads, seq_enc, seq_enc)`) –

Returns The encoder output.

Return type `torch.Tensor` of shape `(batch, seq_enc, dim_model)`

1.13.2 Decoder

class model_center.layer.**Decoder**(*num_layers: int*, *dim_model: int*, *dim_ff: int*, *num_heads: int*, *dim_head: int*, *dtype: torch.dtype = torch.float16*, *int8: bool = False*, *norm_init_var: float = 1.0*, *norm_bias: bool = False*, *norm_eps: float = 1e-05*, *att_init_mean: float = 0.0*, *att_init_std: float = 0.02*, *att_bias: bool = False*, *att_mask_value: float = - inf*, *ffn_init_mean: float = 0.0*, *ffn_init_std: float = 0.02*, *ffn_bias: bool = False*, *ffn_activate_fn: str = 'gated_gelu'*, *pos_bias_type: str = 'none'*, *length_scale: bool = False*, *attn_scale: bool = False*, *dropout_p: float = 0*, *parallel_ffn: bool = False*)

Bases: `torch.nn.modules.module.Module`

Layers of decoder transformer blocks plus an final layernorm.

Parameters

- **num_layers** (*int*) – number of layers.
- **dim_model** (*int*) – main dimension of modules in transformer blocks.
- **dim_ff** (*int*) – dim_ff used in *model_center.layer.FeedForward*.
- **num_heads** (*int*) – num_heads used in *model_center.layer.Attention*.
- **dim_head** (*int*) – dim_head used in *model_center.layer.Attention*.
- **dtype** (*optional*) – Defaults to torch.half.
- **norm_init_var** (*float, optional*) – init_var used in *model_center.layer.LayerNorm*. Defaults to 1.0.
- **norm_bias** (*bool, optional*) – bias used in *model_center.layer.LayerNorm*. Defaults to False.
- **norm_eps** (*float, optional*) – eps used in *model_center.layer.LayerNorm*. Defaults to 1e-5.
- **att_init_mean** (*float, optional*) – init_mean used in *model_center.layer.Attention*. Defaults to 0.0.
- **att_init_std** (*float, optional*) – init_std used in *model_center.layer.Attention*. Defaults to 0.02.
- **att_bias** (*bool, optional*) – bias used in in *model_center.layer.Attention*. Defaults to False.
- **att_mask_value** (*float, optional*) – mask_value used in in *model_center.layer.Attention*. Defaults to float("-inf").
- **ffn_init_mean** (*float, optional*) – init_mean used in *model_center.layer.FeedForward*. Defaults to 0.0.
- **ffn_init_std** (*float, optional*) – init_std used in *model_center.layer.FeedForward*. Defaults to 0.02.
- **ffn_bias** (*bool, optional*) – bias used in *model_center.layer.FeedForward*. Defaults to False.

- **ffn_activate_fn** (*str, optional*) – activate_fn used in *model_center.layer.FeedForward*. Defaults to "gated_gelu".
- **pos_bias_type** (*str, optional*) – pos_bias_type used in *model_center.layer.Attention*. Defaults to "none".
- **post_layer_norm** (*bool, optional*) – whether to use post-layernorm. Defaults to False, which means pre-layernorm.
- **attn_scale** (*bool, optional*) – attn_scale used in in *model_center.layer.Attention*. Defaults to False.
- **dropout_p** (*float, optional*) – Defaults to 0.

forward(*hidden_states: torch.Tensor*, *attention_mask: torch.Tensor*, *position_bias: torch.Tensor*, *cross_hidden_states=None*, *cross_attention_mask=None*, *cross_position_bias=None*)

Parameters

- **hidden_states** (`torch.Tensor` of shape `(batch, seq_dec, dim_model)`) – Input of decoder, Can be the embedding of a batch of sequences.
- **attention_mask** (`torch.Tensor` of shape `(batch, seq_dec, seq_dec)`) – Avoid invalid areas to participate in the calculation.
- **position_bias** (`torch.Tensor` of shape `(num_heads, seq_dec, seq_dec)`) –
- **cross_hidden_states** (`torch.Tensor` of shape `(batch, seq_enc, dim_model)`) – Input of decoder, Can be the output of encoder.
- **cross_attention_mask** (`torch.Tensor` of shape `(batch, seq_dec, seq_enc)`) – Avoid invalid areas to participate in the calculation when the output of encoder participates in the calculation.
- **cross_position_bias** (`torch.Tensor` of shape `(num_heads, seq_dec, seq_enc)`) –

Returns The decoder output.

Return type `torch.Tensor` of shape `(batch, seq_dec, dim_model)`

1.13.3 TransformerBlock

class model_center.layer.**TransformerBlock**(*dim_model: int*, *dim_ff: int*, *num_heads: int*, *dim_head: int*, *is_decoder: bool = False*, *dtype=torch.float16*, *int8=False*, *norm_init_var: float = 1.0*, *norm_bias: bool = False*, *norm_eps: float = 1e-05*, *att_init_mean: float = 0.0*, *att_init_std: float = 0.02*, *att_bias: bool = False*, *att_mask_value: float = - inf*, *ffn_init_mean: float = 0.0*, *ffn_init_std: float = 0.02*, *ffn_bias: bool = False*, *ffn_activate_fn: str = 'gated_gelu'*, *pos_bias_type: str = 'none'*, *post_layer_norm: bool = False*, *parallel_ffn: bool = False*, *length_scale: bool = False*, *attn_scale: bool = False*, *dropout_p: float = 0*)

Bases: `torch.nn.modules.module.Module`

The whole transformer block. A sequence of operation. Consists of self-attention block[, cross-attention block] and feed-forward block.

Parameters

- **dim_model** (*int*) – main dimension of modules in transformer blocks.
- **dim_ff** (*int*) – dim_ff used in *model_center.layer.FeedForward*.
- **num_heads** (*int*) – num_heads used in *model_center.layer.Attention*.
- **dim_head** (*int*) – dim_head used in *model_center.layer.Attention*.
- **is_decoder** (*bool, optional*) – whether to use cross-attention. Defaults to False.
- **dtype** (*optional*) – Defaults to torch.half.
- **norm_init_var** (*float, optional*) – init_var used in *model_center.layer.LayerNorm*. Defaults to 1.0.
- **norm_bias** (*bool, optional*) – bias used in *model_center.layer.LayerNorm*. Defaults to False.
- **norm_eps** (*float, optional*) – eps used in *model_center.layer.LayerNorm*. Defaults to 1e-5.
- **att_init_mean** (*float, optional*) – init_mean used in *model_center.layer.Attention*. Defaults to 0.0.
- **att_init_std** (*float, optional*) – init_std used in *model_center.layer.Attention*. Defaults to 0.02.
- **att_bias** (*bool, optional*) – bias used in in *model_center.layer.Attention*. Defaults to False.
- **att_mask_value** (*float, optional*) – mask_value used in in *model_center.layer.Attention*. Defaults to float("-inf").
- **ffn_init_mean** (*float, optional*) – init_mean used in *model_center.layer.FeedForward*. Defaults to 0.0.
- **ffn_init_std** (*float, optional*) – init_std used in *model_center.layer.FeedForward*. Defaults to 0.02.
- **ffn_bias** (*bool, optional*) – bias used in *model_center.layer.FeedForward*. Defaults to False.
- **ffn_activate_fn** (*str, optional*) – activate_fn used in *model_center.layer.FeedForward*. Defaults to "gated_gelu".
- **pos_bias_type** (*str, optional*) – pos_bias_type used in *model_center.layer.Attention*. Defaults to "none".
- **post_layer_norm** (*bool, optional*) – whether to use post-layernorm. Defaults to False, which means pre-layernorm.
- **attn_scale** (*bool, optional*) – attn_scale used in in *model_center.layer.Attention*. Defaults to False.
- **dropout_p** (*float, optional*) – Defaults to 0.

forward(*self_hidden_states: torch.Tensor*, *self_attention_mask: torch.Tensor*, *self_position_bias: Optional[torch.Tensor] = None*, *cross_hidden_states=None*, *cross_attention_mask=None*, *cross_position_bias=None*)

Parameters

- **self_hidden_states** (torch.Tensor of shape (batch, seq_self, dim_model)) – Input of transformer block(self-attention block). It can be the raw embedding of a batch of sequences.

- **self_attention_mask** (torch.Tensor of shape (batch, seq_self, seq_self)) – Avoid invalid areas to participate in the calculation of self-attention.
- **self_position_bias** (torch.Tensor of shape (num_heads, seq_self, seq_self)) – Provide positional information to self-attention block.
- **cross_hidden_states** (torch.Tensor of shape (batch, seq_cross, dim_model)) – Input of cross-attention block.
- **cross_attention_mask** (torch.Tensor of shape (batch, seq_self, seq_cross)) – Avoid invalid areas to participate in the calculation of cross-attention.
- **cross_position_bias** (torch.Tensor of shape (num_heads, seq_self, seq_cross)) – Provide positional information to cross-attention block.

Returns The output of transformer block.

Return type torch.Tensor of shape (batch, seq_self, dim_model)

1.13.4 FFNBlock

class model_center.layer.**FFNBlock**(*dim_model: int*, *dim_ff: int*, *dtype=torch.float16*, *int8=False*, *norm_init_var: float = 1.0*, *norm_bias: bool = False*, *norm_eps: float = 1e-05*, *ffn_init_mean: float = 0.0*, *ffn_init_std: float = 0.02*, *ffn_bias: bool = False*, *ffn_activate_fn: str = 'gated_gelu'*, *post_layer_norm: bool = False*, *length_scale: bool = False*, *dropout_p: float = 0*)

Bases: torch.nn.modules.module.Module

The whole feed-forward block. A sequence of operation. Consists of layernorm, feed-forward and residual connection.

Parameters

- **dim_model** (*int*) – main dimension of modules in transformer blocks.
- **dim_ff** (*int*) – dim_ff used in *model_center.layer.FeedForward*.
- **dtype** (*optional*) – Defaults to torch.half.
- **norm_init_var** (*float, optional*) – init_var used in *model_center.layer.LayerNorm*. Defaults to 1.0.
- **norm_bias** (*bool, optional*) – bias used in *model_center.layer.LayerNorm*. Defaults to False.
- **norm_eps** (*float, optional*) – eps used in *model_center.layer.LayerNorm*. Defaults to 1e-5.
- **ffn_init_mean** (*float, optional*) – init_mean used in *model_center.layer.FeedForward*. Defaults to 0.0.
- **ffn_init_std** (*float, optional*) – init_std used in *model_center.layer.FeedForward*. Defaults to 0.02.
- **ffn_bias** (*bool, optional*) – bias used in *model_center.layer.FeedForward*. Defaults to False.
- **ffn_activate_fn** (*str, optional*) – activate_fn used in *model_center.layer.FeedForward*. Defaults to "gated_gelu".
- **post_layer_norm** (*bool, optional*) – whether to use post-layernorm. Defaults to False, which means pre-layernorm.

- **dropout_p** (*float, optional*) – Defaults to 0.

forward(*hidden_states: torch.Tensor*)

Parameters **hidden_states** (`torch.Tensor` of shape `(batch, seq_self, dim_model)`) – Hidden states before feed forward layer.

Returns The output of feed-forward block

Return type `torch.Tensor` of shape `(batch, seq_self, dim_model)`

1.13.5 SelfAttentionBlock

class model_center.layer.**SelfAttentionBlock**(*dim_model: int*, *num_heads: int*, *dim_head: int*, *dtype=torch.float16*, *int8=False*, *norm_init_var: float = 1.0*, *norm_bias: bool = False*, *norm_eps: float = 1e-05*, *att_init_mean: float = 0.0*, *att_init_std: float = 0.02*, *att_bias: bool = False*, *att_mask_value: float = - inf*, *pos_bias_type: str = 'none'*, *post_layer_norm: bool = False*, *length_scale: bool = False*, *attn_scale: bool = False*, *dropout_p: float = 0*)

Bases: `torch.nn.modules.module.Module`

The whole cross-attention block. A sequence of operation. Consists of layernorm, self-attention and residual connection.

Parameters

- **dim_model** (*int*) – main dimension of modules in transformer blocks.
- **num_heads** (*int*) – num_heads used in *model_center.layer.Attention*.
- **dim_head** (*int*) – dim_head used in *model_center.layer.Attention*.
- **dtype** (*optional*) – Defaults to torch.half.
- **norm_init_var** (*float, optional*) – init_var used in *model_center.layer.LayerNorm*. Defaults to 1.0.
- **norm_bias** (*bool, optional*) – bias used in *model_center.layer.LayerNorm*. Defaults to False.
- **norm_eps** (*float, optional*) – eps used in *model_center.layer.LayerNorm*. Defaults to 1e-5.
- **att_init_mean** (*float, optional*) – init_mean used in *model_center.layer.Attention*. Defaults to 0.0.
- **att_init_std** (*float, optional*) – init_std used in *model_center.layer.Attention*. Defaults to 0.02.
- **att_bias** (*bool, optional*) – bias used in in *model_center.layer.Attention*. Defaults to False.
- **att_mask_value** (*float, optional*) – mask_value used in in *model_center.layer.Attention*. Defaults to float("-inf").
- **pos_bias_type** (*str, optional*) – pos_bias_type used in *model_center.layer.Attention*. Defaults to "none".
- **post_layer_norm** (*bool, optional*) – whether to use post-layernorm. Defaults to False, which means pre-layernorm.

- **attn_scale** (*bool, optional*) – attn_scale used in in *model_center.layer.Attention*. Defaults to False.
- **dropout_p** (*float, optional*) – Defaults to 0.

forward(*hidden_states: torch.Tensor*, *attention_mask: torch.Tensor*, *position_bias: Optional[torch.Tensor] = None*)

Parameters

- **hidden_states** (`torch.Tensor` of shape `(batch, seq_self, dim_model)`) – Input of self-attention block. It can be the embedding of a batch of sequences.
- **attention_mask** (`torch.Tensor` of shape `(batch, seq_self, seq_self)`) – Avoid invalid areas to participate in the calculation.
- **position_bias** (`torch.Tensor` of shape `(num_heads, seq_self, seq_self)`) – Provide positional information to self-attention block.

Returns The output of attention block.

Return type `torch.Tensor` of shape `(batch, seq_self, dim_model)`

1.13.6 CrossAttentionBlock

class model_center.layer.**CrossAttentionBlock**(*dim_model: int*, *num_heads: int*, *dim_head: int*, *dtype=torch.float16*, *int8=False*, *norm_init_var: float = 1.0*, *norm_bias: bool = False*, *norm_eps: float = 1e-05*, *att_init_mean: float = 0.0*, *att_init_std: float = 0.02*, *att_bias: bool = False*, *att_mask_value: float = - inf*, *pos_bias_type: str = 'none'*, *post_layer_norm: bool = False*, *length_scale: bool = False*, *attn_scale: bool = False*, *dropout_p: float = 0*)

Bases: `torch.nn.modules.module.Module`

The whole cross-attention block. A sequence of operation. Consists of layernorm, cross-attention and residual connection.

Parameters

- **dim_model** (*int*) – main dimension of modules in transformer blocks.
- **num_heads** (*int*) – num_heads used in *model_center.layer.Attention*.
- **dim_head** (*int*) – dim_head used in *model_center.layer.Attention*.
- **dtype** (*optional*) – Defaults to torch.half.
- **norm_init_var** (*float, optional*) – init_var used in *model_center.layer.LayerNorm*. Defaults to 1.0.
- **norm_bias** (*bool, optional*) – bias used in *model_center.layer.LayerNorm*. Defaults to False.
- **norm_eps** (*float, optional*) – eps used in *model_center.layer.LayerNorm*. Defaults to 1e-5.
- **att_init_mean** (*float, optional*) – init_mean used in *model_center.layer.Attention*. Defaults to 0.0.
- **att_init_std** (*float, optional*) – init_std used in *model_center.layer.Attention*. Defaults to 0.02.

- **att_bias** (*bool, optional*) – bias used in in *model_center.layer.Attention*. Defaults to False.
- **att_mask_value** (*float, optional*) – mask_value used in in *model_center.layer.Attention*. Defaults to float("-inf").
- **pos_bias_type** (*str, optional*) – pos_bias_type used in *model_center.layer.Attention*. Defaults to "none".
- **post_layer_norm** (*bool, optional*) – whether to use post-layernorm. Defaults to False, which means pre-layernorm.
- **attn_scale** (*bool, optional*) – attn_scale used in in *model_center.layer.Attention*. Defaults to False.
- **dropout_p** (*float, optional*) – Defaults to 0.

forward(*hidden_states: torch.Tensor*, *key_value_states: torch.Tensor*, *attention_mask: torch.Tensor*, *position_bias: Optional[torch.Tensor] = None*)

Parameters

- **hidden_states** (`torch.Tensor` of shape `(batch, seq_self, dim_model)`) – Input of cross-attention block. It can be seen as query in the coming self-attention operation.
- **key_value_states** (`torch.Tensor` of shape `(batch, seq_cross, dim_model)`) – Used as key_value in coming self_attention operation.
- **attention_mask** (`torch.Tensor` of shape `(batch, seq_self, seq_cross)`) – Avoid invalid areas to participate in the calculation.
- **position_bias** (`torch.Tensor` of shape `(num_heads, seq_self, seq_cross)`) – Provide positional information to self-attention block.

Returns The output of cross-attention block.

Return type `torch.Tensor` of shape `(batch, seq_self, dim_model)`

1.14 Indices and tables

- genindex

INDEX

R

S

T