BMCook

Release 0.1

OpenBMB

Oct 10, 2022

GETTING STARTED

BMCook is a model compression toolkit for large-scale pre-trained language models (PLMs), which integrates multiple model compression methods. You can combine them in any way to achieve the desired speedup.

Note: This project is under active development.

CHAPTER

ONE

CONTENTS

1.1 Introduction

BMCook is a model compression toolkit for large-scale pre-trained language models (PLMs), which integrates multiple model compression methods. You can combine them in any way to achieve the desired speedup. Specifically, we implement the following four model compression methods:

1.1.1 Model Quantization

Quantization compresses neural networks into smaller sizes by representing parameters with low-bit precision, e.g., 8-bit integers (INT8) instead of 32-bit floating points (FP32). It reduces memory footprint by storing parameters in low precision and accelerating the computation in low precision. In this toolkit, we target quantization-aware training, which simulates the computation in low precision during training to make the model parameter adapt to low precision.

In this toolkit, we quantize all linear transformations in the model, which cover over 90% of Transformer computation. For token embedding and attention matrices, we still use floating points, which ensure good performance with little computation cost.

1.1.2 Model Pruning

Pruning compresses neural networks by removing unimportant parameters. According to the granularity of pruning, it is categorized into structured pruning and unstructured pruning. In this toolkit, we implement both pruning methods.

For unstructured pruning, we implement 2:4 sparsity patterns to utilize the TensorCore of NVIDIA GPUs, which brings 1x speedup.

1.1.3 Knowledge Distillation

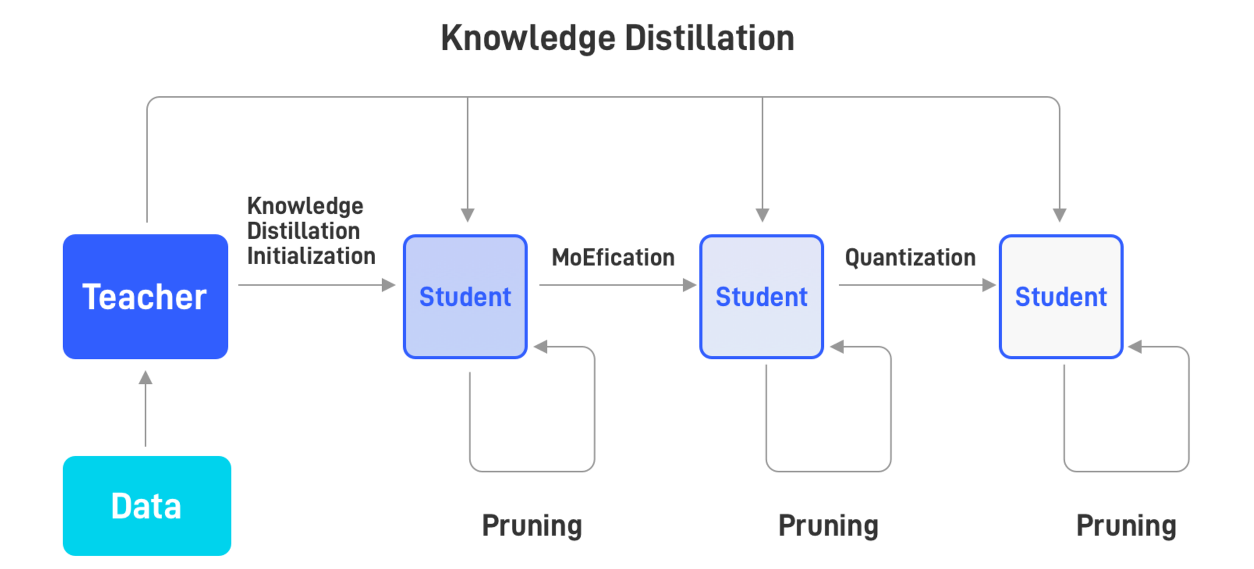
Knowledge distillation aims to alleviate the performance degradation caused by model compression. It provides a more informative training objective than the conventional pre-training does.

In this toolkit, we implement the knowledge distillation losses based on output logits, intermediate hidden states, and attention matrices. In practice, we find that the loss based on output logits is enough.

1.1.4 Model MoEfication

MoEfication utilizes the sparse activation phenomenon in PLMs and splits the feed-forward networks into several small expert networks for conditional computation of PLMs. Please refer to this paper for more details.

1.1.5 Overall Framework



1.2 Installation

To use BMCook, first install BMTrain.

From PyPI (Recommend)

```
$ pip install bmtrain
```

From Source

```
$ git clone https://github.com/OpenBMB/BMTrain.git
$ cd BMTrain
$ python3 setup.py install
```

Please refer to the installation guide of BMTrain for more details.

Then, clone the repository from our github page (don't forget to star us!)

```
$ git clone git@github.com:OpenBMB/BMCook.git
```

In the future, we will support installation from PyPI and provide a one-line model compression API.

1.3 Quick Start

The target of BMCook is to provide a simple way to create a compression script for a pre-trained model. Specifically, if you have a pre-training script, you can introduce compression modules with a few lines of code. You don't need to change the code of model initialization, model loading, and training loop. Just add the following code before the training loop.

1.3.1 Usage of Different Modules

Configuration

The compression strategy is defined in the configuration file. Here is an example of the configuration file:

```
{
    "distillation": {
        "ce_scale": 0,
        "mse_hidn_scale": 1,
        "mse_hidn_module": ["[post]encoder.output_layernorm:[post]encoder.output_
↪layernorm", "[post]decoder.output_layernorm:[post]decoder.output_layernorm"],
        "mse_hidn_proj": false
    },
    "pruning": {
        "is_pruning": true, "pruning_mask_path": "prune_mask.bin",
        "pruned_module": ["ffn.ffn.w_in.w.weight", "ffn.ffn.w_out.weight", "input_
↪embedding"],
        "mask_method": "m4n2_1d"
    },
    "quantization": { "is_quant": true},
    "MoEfication": {
        "is_moefy": false,
        "first_FFN_module": ["ffn.layernorm_before_ffn"]
    }
}
```

Please refer to the API documentation for the detailed explanation of each parameter.

Quantization

You can use `BMQuant` to enable quantization-aware training as follows:

```
BMQuant.quantize(model, config)
```

It will replace all linear modules in the model with quantization-aware modules.

Knowledge Distillation

You can use `BMDistill` to enable knowledge distillation as follows:

```
BMDistill.set_forward(model, teacher_model, foward_fn, config)
```

It will modify the forward function to add distillation loss.

Here is an example of the forward function.

```
def forward(model, enc_input, enc_length, dec_input, dec_length, targets, loss_func,
            output_hidden_states=False):
    outputs = model(
        enc_input, enc_length, dec_input, dec_length, output_hidden_states=output_
↪hidden_states)
    logits = outputs[0]
    batch, seq_len, vocab_out_size = logits.size()

    loss = loss_func(logits.view(batch * seq_len, vocab_out_size), targets.view(batch␣
↪* seq_len))

    return (loss,) + outputs
```

The modified forward function will append the distillation loss to outputs.

Weight Pruning

You can use `BMPrune` to enable pruning-aware training as follows:

```
BMPrune.compute_mask(model, config)
BMPrune.set_optim_for_pruning(optimizer)
```

Based on the pruning mask, `BMPrune` will modify the optimizer to ignore the gradients of pruned weights.

MoEfication

You can use `BMMoE` to get the hidden states for MoEfication:

```
BMMoE.get_hidden(model, config, Trainer.forward)
```

Based on the hidden states, you can use MoEfication to get the corresponding MoE model. For more details, please refer to the API documentation.

1.3.2 Examples Based on CPM-Live

In the `cpm_live_example` folder, we provide the example codes based on CPM-Live.

1.4 Distillation

1.4.1 BMDistill

Here is the example configuration for BMDistill:

```
"distillation": {
        "ce_scale": 0,
        "mse_hidn_scale": 1,
        "mse_hidn_module": ["[post]encoder.output_layernorm:[post]encoder.output_
↪layernorm", "[post]decoder.output_layernorm:[post]decoder.output_layernorm"],
        "mse_hidn_proj": false
}
```

Currently, BMCook supports two kinds of distillation objectives, KL divergence between output distributions (turn on when *ce_scale>0*) and mean squared error (MSE) between hidden states (turn on when *mse_hidn_scale>0*). Practitioners need to specify the hidden states used for MSE by *mse_hidn_module*. Meanwhile, the dimensions of the hidden states may be different between teacher and student models. Therefore, the hidden states of the teacher model need to be projected to the same dimension as those of the student model.Practitioners can turn on *mse_hidn_proj* for simple linear projection.

`class distilling.`**`BMDistill`**

BMDistill provide additional training objectives for knowledge distillation, which further improves the performance of compressed models.

`classmethod set_forward`(*student*, *teacher*, *foward_fn*, *config*)

Modify the forward function of the student model to compute additional knowledge distillation loss.

foward_fn should have the following arguments: *foward_fn(model, enc_input, enc_length, dec_input, dec_length, targets, loss_func)*. These arguments are general for existing Transformers. For decoder-only model, *enc_input* and *enc_length* can be set to None. For encoder-only model, *dec_input* and *dec_length* can be set to None. Similarly, *student* and *teacher* models also have the following arguments: *model(enc_input, enc_length, dec_input, dec_length)*.

Parameters

- **`student`** – Student model.
- **`teacher`** – Teacher model.
- **`foward_fn`** – Forward function of the student model.
- **`config`** – ConfigParser object.

Returns

Modified forward function, whose return values are the original return values of *foward_fn* and additional knowledge distillation loss.

`class distilling.`**`get_module_info`**(*info*)

Parse module info. For example, “[post]encoder.output_layernorm” is parsed to {‘name’: ‘encoder.output_layernorm’, ‘type’: ‘post’}, which means the output of the ‘encoder.output_layernorm’ module is used for distillation. Meanwhile, “[pre]encoder.output_layernorm” is parsed to {‘name’: ‘encoder.output_layernorm’, ‘type’: ‘pre’}, which means the input of the ‘encoder.output_layernorm’ module is used for distillation.

Parameters

`info` – Module info.

`distilling.`**`get_module_map`**`(`*module_list*`)`

Get the module mapping from the teacher model to the student model. For example, "[post]encoder.output_layernorm:[post]encoder.output_layernorm" means that the output of the 'encoder.output_layernorm' module in the teacher model is corresponding to the output of the 'encoder.output_layernorm' module in the student model. The first module name is from the student model, and the second module name is from the teacher model.

Parameters

`module_list` – List of module info.

`distilling.`**`update_forward`**`(`*student*, *teacher*, *s_module_map*, *t_module_map*`)`

Update the forward function of target modules in the student and teacher models.

Parameters

- **`student`** – Student model.
- **`teacher`** – Teacher model.
- **`s_module_map`** – Module mapping from the student model to the teacher model.
- **`t_module_map`** – Module mapping from the teacher model to the student model.

1.5 Pruning

1.5.1 BMPrune

Here is the example configuration for BMPrune:

```
"pruning": {
   "is_pruning": true, "pruning_mask_path": "prune_mask.bin",
   "pruned_module": ["ffn.ffn.w_in.w.weight", "ffn.ffn.w_out.weight", "input_embedding"],
   "mask_method": "m4n2_1d"
}
```

Practitioners can turn on pruning by *is_pruning*. The pruning mask is stored in *pruning_mask_path*. The pruned modules are specified by *pruned_module*. To simplify the list, practitioners can only provide the suffix of the modules. The mask method *mask_method*` is to choose the algorithm for the computation of the pruning mask.

1.6 Quantization

1.6.1 BMQuant

`class quant.`**`BMQuant`**

BMQuant enables quantization-aware training of PLMs by using *cpm-kernels*.

`classmethod` **`quantize`**`(`*model*, *config*`)`

Practitioners can turn on quantization by *is_quant* in the config, which will replace all linear layers with quantized linear layers. BMCook provides the simulation of 8-bit quantization.

Parameters

- **`model`** – Model to quantize.
- **`config`** – Configuration of the quantization.

1.7 MoEfication

1.7.1 BMMoE

To use this module, you need to implement router operation in FFNs as follows:

```
if self.moe is not None:
with torch.no_grad():
      xx_ = input.float().transpose(1,2).reshape(-1, hidden_size)
      xx = xx_ / torch.norm(xx_, dim=-1).unsqueeze(-1)

      score = self.markers(xx)
      labels = torch.topk(score, k=self.k, dim=-1)[1].reshape(bsz, seq_len, self.k)
      cur_mask = torch.nn.functional.embedding(labels, self.patterns).sum(-2).
↪transpose(1,2).detach()
```

```
if self.moe is not None:
   inter_hidden[cur_mask == False] = 0
```

class moe.**BMMoE**

BMMoE replaces the feed-forward modules in PLMs with MoE simulation modules.

static get_hidden(*model*, *config*, *forward_fn*)

Get the hidden states of the model.

foward_fn should have the following arguments: *foward_fn(model, enc_input, enc_length, dec_input, dec_length, targets, loss_func)*. These arguments are general for existing Transformers. For decoder-only model, *enc_input* and *enc_length* can be set to None. For encoder-only model, *dec_input* and *dec_length* can be set to None. Similarly, *student* and *teacher* models also have the following arguments: *model(enc_input, enc_length, dec_input, dec_length)*.

Parameters

- **model** – Model to get the hidden states.
- **config** – Configuration of getting the hidden states. It should contain the names of the layernorm modules before MoEfied FFNs.
- **forward_fn** – Forward function.

INDEX