Phase 3

SYSTEMS FOR MACHINE LEARNING

EE 599

Authors:

Hengyi Tang (8016801929)
Caoyi Zou (2234578518)

Contents

1 Phase 3 - LLaMA2 Model Training

1.1 Initial Setup

1.2 End-To-End Instruction Tuning Flow

Create a file named `finetuning.py`, shown in folder.

1.3 Training Iteration Loop

Replace the HuggingFace `trainer` with the Alpaca repo.

1.4 Gradient Accumulation and Mixed Precision Training

Refering to the Automatic Mixed Precision Recipe and Examples, we implement the following coding. Please refer to `finetuning.py`.

```python
def train():

    # prepare optimizer and loss function
    optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
    criterion = torch.nn.CrossEntropyLoss(ignore_index=IGNORE_INDEX)

    model.train()

    scaler = torch.cuda.amp.GradScaler()

    for epoch in range(5):
        for i, batch in enumerate(dataloader):
            input_ids = batch['input_ids'].to("cuda")
            labels = batch['labels'].to("cuda")

            # 6. gradient checkpointing
            logits = cp.checkpoint(model, input_ids, use_reentrant=False) if CP else model(input_ids)

            # 4. mixed precision training
            with torch.autocast(enabled=MP, device_type='cuda', dtype=torch.float16):
                shift_logits = logits[..., :-1, :].contiguous()
                shift_labels = labels[..., 1:].contiguous()
                shift_logits = shift_logits.view(-1, 32000)
                shift_labels = shift_labels.view(-1)
                loss = criterion(shift_logits, shift_labels)

            scaler.scale(loss).backward()

            # 4. accumulates scaled gradients
            if GA:
                loss = loss / gradient_accumulation_step

                if ((i + 1) % gradient_accumulation_step == 0) or (i + 1 == len(dataloader)):
                    scaler.step(optimizer)
                    scaler.update()
                    optimizer.zero_grad()

            else:
                scaler.step(optimizer)
                scaler.update()
                optimizer.zero_grad()

            print(loss.item())

if __name__ == "__main__":
    train()
```

Figure 1: Step 4&6: Gradient Accumulation, Mixed Precision Training; Gradient Check-pointing

By implementing `autocast` function, which automatically chooses the precision for GPU operations to improve performance while maintaining accuracy, we can use such context managers that allow regions of your script to run in mixed precision.

While setting the if condition, we only update the gradient at some specific check-point, which realize the function of Gradient Accumulation.

```python
# Hyperparameters
GA = True
MP = True
CP = True
LoRA = True
layer_number = 16
sample_number = 200
learning_rate = 1e-5
batch_size = 1
gradient_accumulation_step = 8
```

Figure 2: Hyperparameters and Global Variables

These functions are only valid when the status is set to $True$ of the corresponding hyperparameters, as shown in the above Fig. 2.

1.5 LoRA Linear Layer Module

Implement the LoRA Linear linear layer module. Please refer to the modification in `model.py`.

```python
class Attention(nn.Module):
    """Multi-head attention module."""
    def __init__(self, args: ModelArgs):
        """
        Initialize the Attention module.

        Args:
            args (ModelArgs): Model configuration parameters.

        Attributes:
            n_kv_heads (int): Number of key and value heads.
            n_local_heads (int): Number of local query heads.
            n_local_kv_heads (int): Number of local key and value heads.
            n_rep (int): Number of repetitions for local heads.
            head_dim (int): Dimension size of each attention head.
            wq (ColumnParallelLinear): Linear transformation for queries.
            wk (ColumnParallelLinear): Linear transformation for keys.
            wv (ColumnParallelLinear): Linear transformation for values.
            wo (RowParallelLinear): Linear transformation for output.
            cache_k (torch.Tensor): Cached keys for attention.
            cache_v (torch.Tensor): Cached values for attention.

        """
        super().__init__()
        self.n_kv_heads = args.n_heads if args.n_kv_heads is None else args.n_kv_heads
        self.n_local_heads = args.n_heads
        self.n_local_kv_heads = self.n_kv_heads
        self.n_rep = self.n_local_heads // self.n_local_kv_heads
        self.head_dim = args.dim // args.n_heads

        self.wq = nn.Linear(args.dim, args.n_heads * self.head_dim, bias=False)
        self.wk = nn.Linear(args.dim, self.n_kv_heads * self.head_dim, bias=False)
        self.wv = nn.Linear(args.dim, self.n_kv_heads * self.head_dim, bias=False)
        self.wo = nn.Linear(args.n_heads * self.head_dim, args.dim, bias=False)

        # Initialize LoRA layer
        if LORA:
            self.lora_layer = LoRA(args.dim, args.dim, r, alpha, dropout_rate)

```

Figure 3: Initialization of LoRA layer

Implement the initialization of the LoRA linear layer in `class Attention`, and deploy the LoRA model from file `lora.py`. Both the input and output feature number is set to `args.dim`, in order to coordinate to the original weights size.

```python
class Attention(nn.Module):
    def forward(

        # repeat k/v heads if n_kv_heads < n_heads
        keys = repeat_kv(xk, self.n_rep)  # (bs, cache_len + seqlen, n_local_heads, head_dim)
        values = repeat_kv(xv, self.n_rep)  # (bs, cache_len + seqlen, n_local_heads, head_dim)

        xq = xq.transpose(1, 2)  # (bs, n_local_heads, seqlen, head_dim)
        keys = keys.transpose(1, 2) # (bs, n_local_heads, cache_len + seqlen, head_dim)
        values = values.transpose(1, 2) # (bs, n_local_heads, cache_len + seqlen, head_dim)
        scores = torch.matmul(xq, keys.transpose(2, 3)) / math.sqrt(self.head_dim)
        if mask is not None:
            scores = scores + mask  # (bs, n_local_heads, seqlen, cache_len + seqlen)
        scores = F.softmax(scores.float(), dim=-1).type_as(xq)
        output = torch.matmul(scores, values)  # (bs, n_local_heads, seqlen, head_dim)
        output = output.transpose(1, 2).contiguous().view(bsz, seqlen, -1)

        # Using LoRA
        if LORA and hasattr(self, 'lora_layer') and self.lora_layer is not None:
            output = self.lora_layer(output)

        return self.wo(output)
```

Figure 4: Output Tranfer in the LoRA layer

During the forward propagation process of `Attention` function, if LoRA is deployed, then transfer the output with certain model.

```python
# Hyperparameters:
LORA = True
r = 16
alpha = 32
dropout_rate = 0.05
```

Figure 5: Hyperparameters for LoRA Model

1.6 Gradient Check-pointing

For this process, we implement one capsulation formula of check-pointing, as shown in Fig. 1.

1.7 Model Fine-Tuning

Also, in the file `finetuning.py`, encompass the model training part with the variables to calculate the training time and the peak RAM memory usage, print them for further data analysis. Finally, save the LoRA parameters. Shown in Fig. 6 and Fig. 7.

```python
    start_time = time.time()
    torch.cuda.reset_peak_memory_stats()

```

Figure 6: Print Training Time

```python
    end_time = time.time()
    total_time = end_time - start_time
    print(f"Total training time: {total_time*4} seconds")

    peak_memory = torch.cuda.max_memory_allocated() / (1024 ** 2)
    print(f"Peak memory usage during training: {peak_memory:.2f} MB")

    # save LoRA weights
    if GA and MP and LoRA and CP:
        model_weights = model.state_dict()
        lora_weights = {k: v for k, v in model_weights.items() if "lora_" in k}
        torch.save(lora_weights, "lora_weights.pth")

if __name__ == "__main__":
    train()
```

Figure 7: Print Training Time and Memory Usage

Load the just saved Lora parameters in `inference.py` for text generation. Shown in Fig. 8

```python

    # LoRA weights
    lora_weights = torch.load(lora_weights_path, map_location="cpu")
    for name, param in model.named_parameters():
        if name in lora_weights:
            param.data.copy_(lora_weights[name])

```

Figure 8: Load LoRA Parameters

In order to extract the first 200 samples of the database, modify function, `SuperviseDataset` and `make_supervised_data_module` in the file `model.py`, and class to limit the volumn of data loaded. Refer to the `model.py`, shown in Fig. 9 and Fig. 10.

```python
class SupervisedDataset(Dataset):
    """Dataset for supervised fine-tuning."""

    def __init__(self, data_path, tokenizer, num_samples=None):
        super(SupervisedDataset, self).__init__()
        logging.warning("Loading data...")
        with open(data_path, "r") as f:
            list_data_dict = json.load(f)

        logging.warning("Formatting inputs...")
        prompt_input, prompt_no_input = PROMPT_DICT["prompt_input"], PROMPT_DICT["prompt_no_input"]
        sources = [
            prompt_input.format_map(example) if example.get("input", "") != "" else prompt_no_input.format_map(example)
            for example in list_data_dict[:num_samples]
        ]
        targets = [f"{example['output']}" for example in list_data_dict[:num_samples]]

        logging.warning("Tokenizing inputs... This may take some time...")
        data_dict = preprocess(sources, targets, tokenizer)

        self.input_ids = data_dict["input_ids"]
        self.labels = data_dict["labels"]
```

Figure 9: Extract database Fcn1

```python
def make_supervised_data_module(tokenizer, data_path, num_samples=None):
    """Make dataset and collator for supervised fine-tuning."""
    train_dataset = SupervisedDataset(tokenizer=tokenizer, data_path=data_path, num_samples=num_samples)
    data_collator = DataCollatorForSupervisedDataset()
    return dict(train_dataset=train_dataset, eval_dataset=None, data_collator=data_collator)
```

Figure 10: Extract database Fcn2

1.8 Hyperparameters

Additionally, set the `global variables` at the top front as assigned. Also set the global variables for the LoRA model.

1.9 Analysis and Measurements

1. System performance analysis

		Grad. Accumulation	Grad. Checkpoint	Mixed Precision	LoRA
Memory	parameter	–	–	↑	↑
	activation	–	↓	↓	–
	gradient	–	–	↓	–
	optimizer state	–	–	–	↑
Computation		↓	↑	↓	↓

Table 1: System performance analysis

For 'Gradient Accumulation' case, the memory allocated for all the variables, ranging from 'parameter', 'activation', 'gradient' and 'optimizer state', will not increase, cause the network structure is not modified. For the 'gradient', since we just update the values on the same address location, thus we do not need extra memory. For the 'optimizer state', we are not applying different optimizers for additional data maintaining for the model parameters. Since we do not need to do the backward propogation for the gradient each time, the computation time will be decreased.

For 'Gradient Check-pointing' case, we don not need additional optimizer for extra data maintaining, thus the memory for 'optimizer state' stays unchanged. Also, the 'parameter' and the 'gradient' for calculating the weights stays still, cause the number of those are determined by the configuration of network, which stays still. For the 'activation, since we only remain the ones at certain check points, and abandon other redundant ones, and only recalculating them when needed, thus reduce certain memory for this. The optimization for the deduction process, however, need the leverage between computation time, which needs to recalculate the discarded activation during the backward propagation, making the computation process less precise and efficient.

For 'Mixed Precision' case, same as 'Gradient Check-pointing' case, the memory for 'optimizer state' stays unchanged. For the 'parameter', it is often necessary to obtain an additional copy of the weights, which is usually full-precision (single precision). This means that the overall weight memory consumption is now 50% higher compared to using full precision weights alone. This is because you need both the original full-precision weights and their half-precision copy. Which is the tradeoff for decreasing the memory needed for the half-precision version 'activation' and 'gradient'. Since we replace the FP32 with FP32, the computation time will be deduced credit to less precision.

For 'LoRA' case, memory consumption stay still for the 'activation' part since the updating structure is not altered. For the 'parameter' and 'gradient', because while the original weight W_o is frozen and does not participate in updates, which means we only need to get another two extra matrix A and B involved in the gradient updating process. While these two matrix need additional memory, augmenting the memory required for storation. And the Computation is also speeded up.

2. System performance measurement

   In this part, we process 200 training samples on the A100 GPU and obtain the following results. Screenshots also attached.

GA	OFF				ON			
MP	OFF		ON		OFF		ON	
LoRA	OFF	ON	OFF	ON	OFF	ON	OFF	ON
Peak Mem	35909	36442	35909	36442	35909	36442	35909	36442
Runtime	176.0	198.4	185.6	193.6	139.2	145.6	140.8	156.8

Table 2: System performance measurement

```
GA: False || MP: False || LoRA: False || CP: True
trainable params: 6,584,252,416 || all params: 6,584,252,416 || trainable%: 100.00
9.30508804321289
8.497645378112793
3.179842948913574
3.166210651397705
0.16252948343753815
1.5253150463104248
0.025018714368343353
0.6202989220619202
0.34347233176231384
0.13388946652412415
Total training time: 11.011384963989258 seconds
Peak memory usage during training: 35909.71 MB
```

Figure 11: Status: GA=OFF / MP=OFF / LoRA=OFF

```
GA: False || MP: False || LoRA: True || CP: True
trainable params: 6,588,037,120 || all params: 7,057,799,168 || trainable%: 93.34
11.470152854919434
12.277411460876465
5.2643961906433105
6.274170398712158
2.5364913940429688
3.9863779544830322
1.4870327711105347
2.748659133911133
1.1614826917648315
2.2827136516571045
Total training time: 12.425705909729004 seconds
Peak memory usage during training: 36442.33 MB
```

Figure 12: Status: GA=OFF / MP=OFF / LoRA=ON

```
GA: False || MP: True || LoRA: False || CP: True
trainable params: 6,584,252,416 || all params: 6,584,252,416 || trainable%: 100.00
9.30508804321289
8.497645378112793
3.179842948913574
3.166210651397705
0.16252948343753815
1.5253150463104248
0.025018714368343353
0.6202989220619202
0.34347233176231384
0.13388946652412415
Total training time: 11.589696884155273 seconds
Peak memory usage during training: 35909.71 MB
```

Figure 13: Status: GA=OFF / MP=ON / LoRA=OFF

```
GA: False || MP: True || LoRA: True || CP: True
trainable params: 6,588,037,120 || all params: 7,057,799,168 || trainable%: 93.34
11.470152854919434
12.277411460876465
5.2643961906433105
6.274170398712158
2.5364913940429688
3.9863779544830322
1.4870327711105347
2.748659133911133
1.1614826917648315
2.2827136516571045
Total training time: 12.15066146850586 seconds
Peak memory usage during training: 36442.33 MB
```

Figure 14: Status: GA=OFF / MP=ON / LoRA=ON

```
GA: True || MP: False || LoRA: False || CP: True
trainable params: 6,584,252,416 || all params: 6,584,252,416 || trainable%: 100.00
1.1631360054016113
1.2016323804855347
0.34876787662506104
0.6877389550209045
0.07325665652751923
0.42632725834846497
0.006354565266519785
0.2554172873497009
0.13230770826339722
0.003307815408334136
Total training time: 8.745233535766602 seconds
Peak memory usage during training: 35909.71 MB
```

Figure 15: Status: GA=ON / MP=OFF / LoRA=OFF

```
GA: True || MP: False || LoRA: True || CP: True
trainable params: 6,588,037,120 || all params: 7,057,799,168 || trainable%: 93.34
1.4337691068649292
1.3924968242645264
1.2617956399917603
1.5308793783187866
0.33667463064193726
0.8056578636169434
0.29767104983329773
0.6969085335731506
0.17832493782043457
0.48961690068244934
Total training time: 9.081843376159668 seconds
Peak memory usage during training: 36442.33 MB
```

Figure 16: Status: GA=ON / MP=OFF / LoRA=ON

```
GA: True || MP: True || LoRA: False || CP: True
trainable params: 6,584,252,416 || all params: 6,584,252,416 || trainable%: 100.00
1.1631360054016113
1.2016323804855347
0.34876787662506104
0.6877389550209045
0.07325665652751923
0.42632725834846497
0.006354565266519785
0.2554172873497009
0.13230770826339722
0.003307815408334136
Total training time: 8.823179244995117 seconds
Peak memory usage during training: 35909.71 MB
```

Figure 17: Status: GA=ON / MP=ON / LoRA=OFF

```
GA: True || MP: True || LoRA: True || CP: True
trainable params: 6,588,037,120 || all params: 7,057,799,168 || trainable%: 93.34
1.4337691068649292
1.3924968242645264
1.2617956399917603
1.5308793783187866
0.33667463064193726
0.8056578636169434
0.29767104983329773
0.6969085335731506
0.17832493782043457
0.48961690068244934
Total training time: 9.82708740234375 seconds
Peak memory usage during training: 36442.33 MB
```

Figure 18: Status: GA=ON / MP=ON / LoRA=ON

1.10 Final Result

Overall, it still outputs text rather than gibberish, but there are spelling and semantic errors.

```
(pyvenv) [hengyita@b02-20 ml-systems-final-project-HelloworldHarley]$ python3 inference.py
/scratch1/hengyita/pyvenv/lib/python3.11/site-packages/torch/__init__.py:696: UserWarning: torch.set_default_tensor_type() is deprecated as of PyTorch 2.1, please use torch.set_default_dtype() and torch.set_default_device() as alternatives. (Triggered internally
 at ../torch/csrc/tensor/python_tensor.cpp:451.)
  _C._set_default_tensor_type(t)
I believe the meaning of life is
> to be happy. sierpina2000@gmail comgmail comaugustina2 энциклопедия life is to be happyaugustina life is to be happyaugustina life is to be happyaugustina life is to be happyaugustina life is to be happyaugustina life is

==================================

Simply put, the theory of relativity states that
> speed and mass play roles in how fast something travels through space time continuum. Einzeln recherchiertes Produkt anzeigen produkt details produkt details produkt details produkt details produkt details produkt details produkt details produkt details prod
ukt details produkt details produkt details produkt details produ

==================================

A brief message congratulating the team on the launch:

        Hi everyone,

        I just
> want to congratulate each one of you on the launch of our newest project called `project name`. Congratulations again on your effort towards this project successfull completion. Congratulations again on your effort towards this project successfull completion.
Congratulations again on your effort towards this project successfull completion

==================================

Translate English to French:

        sea otter => loutre de mer
        peppermint => menthe poivrée
        plush girafe => girafe peluche
        cheese =>
> fromage
        pepperoni pizza => pizza hawaïenne au pepperoni
        banana => banana
        potato => potato
        banana => banana
        potato => potato
        banana => banana
        potato => potato


==================================
```

Figure 19: Final Result