COS 598D Project: Distributed Training of Language Models

Wenzhe Li (wl5915)

Github repo: https://github.com/wenzhe-li/cos598d_sp24

Task 1

Implementation details:

```
# TODO(cos598d): perform backward pass here
loss.backward()

# TODO(cos598d): perform a single optimization step (parameter update) by
invoking the optimizer
optimizer.step()

# TODO(cos598d): call evaluate() here to get the model performance after
every epoch.
evaluate(args, model, tokenizer, prefix="")

# TODO(cos598d): load the model using from_pretrained. Remember to pass in
`config` as an argument.
# If you pass in args.model_name_or_path (e.g. "bert-base-cased"), the
model weights file will be downloaded from HuggingFace.
model = model_class.from_pretrained(args.model_name_or_path, config=config)
```

Loss values of the first five minibatches:

	iter=1	**iter=2**	**iter=3**	**iter=4**	**iter=5**
loss	0.763	0.775	0.716	0.752	0.726

Evaluation metric after every epoch:

	epoch=1	**epoch=2**	**epoch=3**
acc	0.614	0.628	0.653

Task 2(a)

Implementation details:

```
# Distributed sampler
if args.local_rank == -1:
    train_sampler = RandomSampler(train_dataset)
```

```
else:
    train_sampler = DistributedSampler(train_dataset,
num_replicas=args.world_size, rank=args.local_rank)
train_dataloader = DataLoader(train_dataset, sampler=train_sampler,
batch_size=(args.train_batch_size // args.world_size))

# Gather gradients to rank 0 and then scatter the average gradient to all
ranks
for param in model.parameters():
    gathered_gradients = [torch.zeros_like(param.data) for _ in
range(args.world_size)]
    torch.distributed.gather(param.grad, gather_list=gathered_gradients if
args.local_rank == 0 else None, dst=0)
    torch.distributed.barrier()
    if args.local_rank == 0:
        mean_gradient = torch.mean(torch.stack(gathered_gradients), dim=0)
    else:
        mean_gradient = torch.zeros_like(param.grad)
    scatter_list = [mean_gradient for _ in range(args.world_size)] if
args.local_rank == 0 else None
    torch.distributed.scatter(mean_gradient, scatter_list=scatter_list,
src=0)
    torch.distributed.barrier()
    param.grad = mean_gradient
torch.distributed.barrier()

# Initialize the distributed environment
torch.distributed.init_process_group(backend="nccl", init_method="tcp://{}:
{}".format(args.master_ip, args.master_port), world_size=args.world_size,
rank=args.local_rank)
```

Loss values are stored in `loss_{worker_id}.txt`, which are similar to the single-node training case.

Task 2(b)

Implementation details:

```
# Gather gradients to rank 0 and then scatter the average gradient to all
ranks
for param in model.parameters():
    torch.distributed.all_reduce(param.grad,
op=torch.distributed.ReduceOp.SUM)
    param.grad = param.grad / args.world_size
torch.distributed.barrier()
```

Loss values are stored in `loss_{worker_id}.txt`, which are identical to loss values in Task 2(a).

Task 3

Implementation details:

```
# Register the model with DDP
if args.local_rank != -1:
    model = DDP(model, device_ids=[0], output_device=0)
```

Difference among different setups

Average time per iteration (evaluated on 4 GPUs when running other tasks, maybe slower than usual):

	task 2(a)	task 2(b)	task 3
average time per iteration (s)	3.730	0.836	0.438

According to this table, we observe that PyTorch original DistributedDataParallel is faster than gradient synchronization with all_reduce, and both two methods are much faster than gradient synchronization with gather and scatter. Note that this result is applicable to distributed training on multi-GPUs. According the paper mentioned in the README, communication is the dominant training latency contributor in distributed learning. Given that DDP and all_reduce are higher level interfaces than gather and scatter, I believe they have incorporated more clever communication approaches and hence produced faster training speed. Indeed, DDP will overlap the gradient computation with communication, which can save much of the communication latency.

Scalability of distributed ML

I think distributed learning is especially useful for large-scale model training. For instance, some large language models may have a huge amount of parameters, and even the most powerful GPU cannot afford large batch training on it. In this case, we can use data parallel training to pass data through multiple GPUs, and achieve more stable training via larger batch size. Furthermore, the comparison between different data parallel training approaches shows that even though the high-level idea is simple, it takes huge engineering efforts to improve the learning efficiency up to 10 times or more.