Moving Stuff Around

A study on the efficiency of moving documents into memory for Neural IR models

Arthur Câmara - TU Delft/Naver Labs Europe

Let’s train a Neural (re-)Ranker.

Simple enough, right?
A Cross-Encoder with a linear layer over the CLS tokens.

```python
from torch import nn
from transformers import AutoModel

class CrossEncoderModel(nn.Module):
    def __init__(self, config) -> None:
        super().__init__(config)
        self.bert_model = AutoModel.from_pretrained(config._name_or_path)
        self.dropout = nn.Dropout(0.1)
        self.classifier = nn.Linear(config.hidden_size, 1)
        self.loss = nn.BCEWithLogitsLoss()

    def forward(self, input_ids, attention_mask, labels):
        outputs = self.bert_model(input_ids, attention_mask=attention_mask)
        CLS_tokens = outputs.last_hidden_state[:, 0, :]
        pooled_outputs = self.dropout(CLS_tokens)
        logits = self.classifier(pooled_outputs).view(-1)
        loss = self.loss(logits.view(-1), labels).mean()
        return loss, logits
```

How about the data?

```python
class MyDataset(Dataset):
    def __init__(self, docs_path, queries_path, qrels):
        self.all_doc_ids = []
        self.docs = {}
        self.queries = {}
        for line in open(docs_path):
            d_id, doc = line.strip().split("\t", maxsplit=1)
            self.all_doc_ids.append(d_id)
            self.docs[d_id] = doc

        for line in open(queries_path):
            d_id, doc = line.strip().split("\t", maxsplit=1)
            self.queries[d_id] = doc
        self.train_qrels = pytrec_eval.parse_qrel(open(qrels))
        self.q_ids = dict(enumerate(self.train_qrels.keys()))

    def __getitem__(self, index):
        q_id = self.q_ids[index]
        d_id = list(self.train_qrels[q_id].keys())[0]
        neg_id = random.choice(self.all_doc_ids)

        return {"query_text": self.queries[q_id],
            "doc_text": self.docs[d_id],
            "neg_text": self.docs[neg_id],
        }
```

PyTorch’s Dataset makes it so easy!

Training Loop for multiple GPUs?

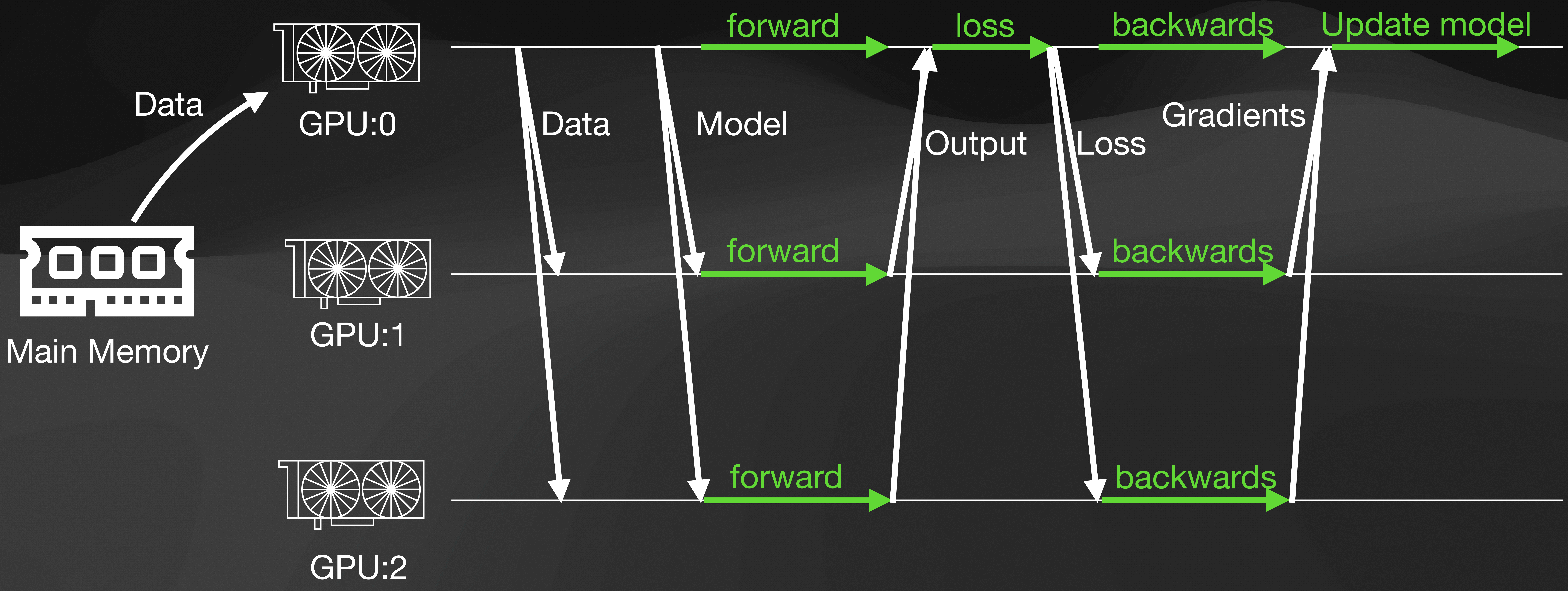
```python
train_dataset = MyDataset("msmarco_docs", "msmarco_queries", "msmarco_qrels")
model_config = AutoConfig.from_pretrained("distilbert-base-uncased")
model = nn.DataParallel(CrossEncoderModel(model_config))
device = torch.device("cuda")
model.to(device)
loader = DataLoader(train_dataset, batch_size=8)
optimizer = transformers.AdamW(model.parameters())
optimizer.zero_grad()
model.train()
for features, labels in loader:
    for k in features.keys():
        features[k] = features[k].to(device)
    labels = labels.to(device)
    loss, _ = model(**features, labels=labels)
    loss = loss.mean(dim=0)
    loss.backward(); optimizer.step(); optimizer.zero_grad()
```

What’s the problem here?

```
train_dataset = MyDataset("msmarco_docs", "msmarco_queries", "msmarco_qrels")
model_config = AutoConfig.from_pretrained("distilbert-base-uncased")
model = nn.DataParallel(CrossEncoderModel(model_config))
device = torch.device("cuda")
model.to(device)
loader = DataLoader(train_dataset, batch_size=8)
optimizer = transformers.AdamW(model.parameters())
optimizer.zero_grad()
model.train()
for features, labels in loader:
    for k in features.keys():
        features[k] = features[k].to(device)
    labels = labels.to(device)
    loss, _ = model(**features, labels=labels)
    loss = loss.mean(dim=0)
    loss.backward(); optimizer.step(); optimizer.zero_grad()
```

What's the problem?

DataParallel



What’s the problem?

- Too much communication overhead!
  - The faster the GPU, the more noticeable the overhead.
  - Link speed also matters! NVLink is faster than PCI!
- Each GPU has its own thread for processing the input.
  - Even if all data goes through GPU:0, pre-processing is multi-threaded.
  - Which can lead to problems with the GIL.

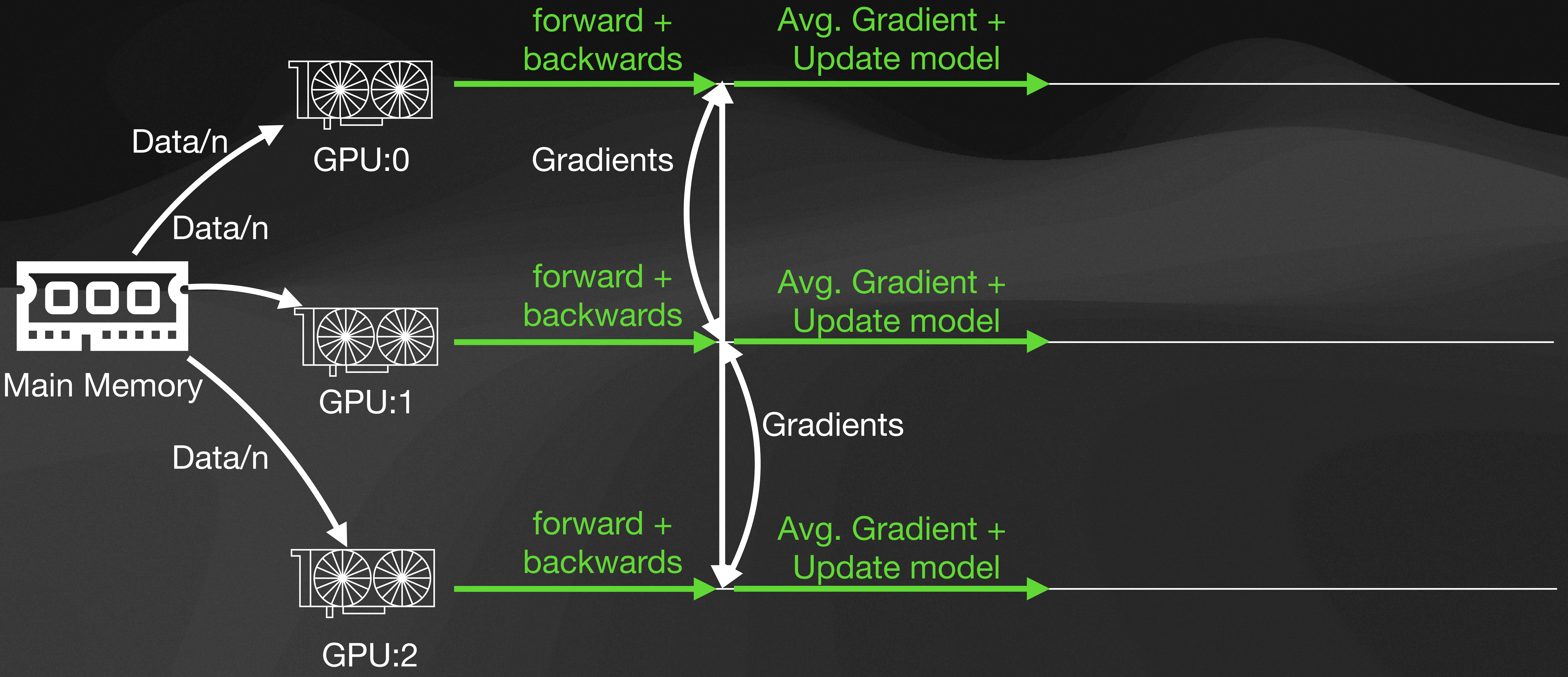
The what now?

Global Interpreter Lock

- DataParallel creates one thread per GPU
- In (C)Python, only ONE Python thread can run at a time.
- The Global Interpreter Lock (**GIL**) is a Mutex lock that enforces that.
- Great for code written in C (or Rust, like tokenizers)
- Not very useful for (almost) anything else that runs in parallel.

An improvement!

DistributedDataParallel



Distributed Data Parallel

- Significantly lower communication overhead!
  - Unless your GPUs are communicating via PCI instead of NVLink.
- One PROCESS per GPU. Completely sidesteps the GIL.
- It can also distribute the MODEL between GPUs, making it possible to train models that wouldn't fit in a single GPU.
- DDP creates one copy of **everything** for each GPU.

Anything else?

```python
class MyDataset(Dataset):
    def __init__(self, docs_path, queries_path, qrels):
        self.all_doc_ids = []
        self.docs = {}
        self.queries = {}
        for line in open(docs_path):
            d_id, doc = line.strip().split("\t", maxsplit=1)
            self.all_doc_ids.append(d_id)
            self.docs[d_id] = doc

        for line in open(queries_path):
            d_id, doc = line.strip().split("\t", maxsplit=1)
            self.queries[d_id] = doc
        self.train_qrels = pytrec_eval.parse_qrel(open(qrels))
        self.q_ids = dict(enumerate(self.train_qrels.keys()))

    def __getitem__(self, index):
        q_id = self.q_ids[index]
        d_id = list(self.train_qrels[q_id].keys())[0]
        neg_id = random.choice(self.all_doc_ids)

        return {"query_text": self.queries[q_id],
            "doc_text": self.docs[d_id],
            "neg_text": self.docs[neg_id],
        }
```

- Each process will have a copy of ALL documents
- MsMarco v1 takes ~20GB
- MsMarco v2, ~100GB.
- ClueWeb22 is coming.
- Unless you have infinite RAM, not a good idea!

Alternatives?

```python
import json

def get_document(document_id):
    (string1, string2, bundlenum, position) = document_id.split('_')
    assert string1 == 'msmarco' and string2 == 'doc'

    with open(f'./msmarco_v2_doc/msmarco_doc_{bundlenum}', 'rt', encoding='utf8') as in_fh:
        in_fh.seek(int(position))
        json_string = in_fh.readline()
        document = json.loads(json_string)
        assert document['docid'] == document_id
        return document

document = get_document('msmarco_doc_31_726131')
print(document.keys())
```

- TREC-DL have an idea:
  - Use a *pointer* to the document.
  - Shrinks memory usage to ~4GB/GPU
- Or, rather, use an existing library, like ir_datasets!
  - Even faster at some times (uses NumPy's memarray)
  - Already have everything you may need (queries, qrels, documents, etc)
- Both options are *FASTER* than loading into memory!

How much faster?

Number of GPUs	1 GPU	2 GPUs	4GPUs	8 GPus
In-memory	39.65 samples/s	73.73 samples/s	OOM	OOM
Indexed (Trec-DL style)	39.71 samples/s	74.98 samples/s	140.04 samples/s	262.62 samples/s
ir_datasets	39.16 samples/s	75.45 samples/s	**141.31 samples/s**	**264.08 samples/s**

- Ignoring “warm-up” time
- GTX 1080Tis, without NVLink
- Using Distributed DataParallel
- 256GB of Memory

- OOM: Out-of-memory
- Teal: Faster option for that number of GPUs
- Code and dashboard here:
    - https://github.com/ArthurCamara/ir_efficiency/

Moving forward

Caveats

- We measured these numbers WITHOUT NVLink, which can increase DDP performance even further.
- Same for Mixed and Half-precision weights. (i.e., FP16). It should also lead to faster results.
- We only measured Transformers and PyTorch.
  - What about TF-Ranking?
- There are plenty of libraries to speed-up training, like ray.io and PyTorch Lightning.
  - We haven’t checked these.
- For **decades**, the GIL is supposedly disappearing. If so, DP should improve.
- We got it (kind of) wrong in the paper. DDP has LOWER overhead than DP if using NvLink.

Take-home message and recommendations

✗ Don’t load all of the documents in memory.

It’s slower, and as datasets grow, it will be unusable on larger datasets.

✗ Don’t use DataParallel.

It has a higher overhead and may lead to problems with the GIL.

✓ Be more conscious of how you read data from disk.

If the dataset is available on ir_datasets, use it. Otherwise, a lookup table can work well.

✓ Do use DistributedDataParallel.

It has less overhead, scales better, and libraries like HF Accelerator implement it for you.