手把手 带你实战 Transformers

你可是处女座啊

基础入门篇

Transformers基础使用指南

你可是处女座啊

目录

01 基础知识与环境安装

（1）自然语言处理任务

（2）Transformers介绍

（3）Transformers相关环境安装

（4）两行代码的QA实例

常见自然语言处理任务

情感分析（sentiment-analysis）：对给定的文本分析其情感极性

文本生成（text-generation）：根据给定的文本进行生成

命名实体识别（ner）：标记句子中的实体

阅读理解（question-answering）：给定上下文与问题，从上下文中抽取答案

掩码填充（fill-mask）：填充给定文本中的掩码词

文本摘要（summarization）：生成一段长文本的摘要

机器翻译（translation）：将文本翻译成另一种语言

特征提取（feature-extraction）：生成给定文本的张量表示

对话机器人（conversional）：根据用户输入文本，产生回应，与用户对话

自然语言处理的几个阶段

- 第一阶段：统计模型 + 数据（特征工程）
  - 决策树、SVM、HMM、CRF、TF-IDF、BOW
- 第二阶段：神经网络 + 数据
  - Linear、CNN、RNN、GRU、LSTM、Transformer、Word2vec、Glove
- 第三阶段：神经网络 + 预训练模型 + （少量）数据
  - GPT、BERT、RoBERTa、ALBERT、BART、T5
- 第四阶段：神经网络 + 更大的预训练模型 + Prompt
  - ChatGPT、Bloom、LLaMA、Alpaca、Vicuna、MOSS、文心一言、通义千问、星火

Transformers简单介绍

- 官方网址：https://huggingface.co/
- HuggingFace出品，当下最热、最常使用的自然语言处理工具包之一，不夸张的说甚至没有之一
- 实现了大量的基于Transformer架构的主流预训练模型，不局限于自然语言处理模型，还包括图像、音频以及多模态的模型
- 提供了海量的预训练模型与数据集，同时支持用户自行传，社区完善，文档全面，三两行代码便可快速实现模型训练推理，上手简单

一句话总结：学就对了！

Transformers及相关库

- Transformers：核心库，模型加载、模型训练、流水线等
- Tokenizer：分词器，对数据进行预处理，文本到token序列的互相转换
- Datasets：数据集库，提供了数据集的加载、处理等方法
- Evaluate：评估函数，提供各种评价指标的计算函数
- PEFT：高效微调模型的库，提供了几种高效微调的方法，小参数量撬动大模型
- Accelerate：分布式训练，提供了分布式训练解决方案，包括大模型的加载与推理解决方案
- Optimum：优化加速库，支持多种后端，如Onnxruntime、OpenVino等
- Gradio：可视化部署库，几行代码快速实现基于Web交互的算法演示系统

Transformers环境安装

前置环境安装——python

- **miniconda 安装**
  - 下载地址：https://mirrors.tuna.tsinghua.edu.cn/anaconda/miniconda/
  - 如果C盘有空间，最好安装在C盘，且安装目录中不能有中文
  - 勾选将其添加到PATH
- **conda环境创建**
  - 命令：conda create -n transformers python=3.9
  - 明确指定版本，否则可能会因版本过高导致有包装不上
- **pypi配置国内源**
  - 清华源：https://mirrors.tuna.tsinghua.edu.cn/help/pypi/

Transformers环境安装

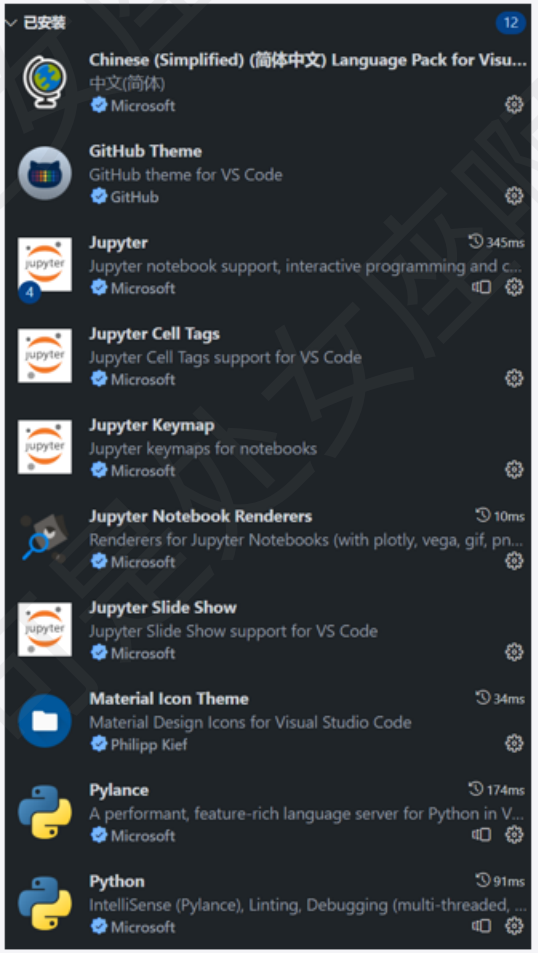
前置环境安装—pytorch

- **pytorch安装**
  - 官方地址：https://pytorch.org/
  - 在一个单独的环境中，能使用pip就尽量使用pip，实在有问题的情况，例如没有合适的编译好的系统版本的安装包，再使用conda进行安装，不要来回混淆
  - 30XX、40XX显卡，要安装cu11以上的版本，否则无法运行
- **CUDA是否要安装**
  - 如果只需要训练、简单推理，则无需单独安装CUDA，直接安装pytorch
  - 如果有部署需求，例如导出TensorRT模型，则需要进行CUDA安装

Transformers环境安装

前置环境安装—vscode

- **VS Code 安装**
  - 官方地址：https://code.visualstudio.com/download
- **插件安装**
  - Python (代码编写)
  - remote ssh (连接服务器)
  - Chinese Language Pack（简体中文包）
- **终端设置（非常重要！非常重要！非常重要！）**
  - 选择默认配置文件: cmd.exe



Transformers环境安装

Transformers安装

- **安装命令**
  - pip install transformers datasets evaluate peft accelerate gradio optimum sentencepiece
  - pip install jupyterlab scikit-learn pandas matplotlib tensorboard nltk rouge
- **hosts修改（暂时用不上了）**
  - 185.199.108.133 raw.githubusercontent.com
  - 185.199.109.133 raw.githubusercontent.com
  - 185.199.110.133 raw.githubusercontent.com
  - 185.199.111.133 raw.githubusercontent.com
  - 2606:50c0:8000::154 raw.githubusercontent.com
  - 2606:50c0:8001::154 raw.githubusercontent.com
  - 2606:50c0:8002::154 raw.githubusercontent.com
  - 2606:50c0:8003::154 raw.githubusercontent.com

Transformers极简实例

三行代码，启动NLP应用

样例1：文本分类

```
# 导入gradio
import gradio as gr
# 导入transformers相关包
from transformers import *
# 通过Interface加载pipeline并启动文本分类服务
gr.Interface.from_pipeline(pipeline("text-classification", model="uer/roberta-base-finetuned-dianping-chinese")).launch()
```

样例2：阅读理解

```
# 导入gradio
import gradio as gr
# 导入transformers相关包
from transformers import *
# 通过Interface加载pipeline并启动阅读理解服务
gr.Interface.from_pipeline(pipeline("question-answering", model="uer/roberta-base-chinese-extractive-qa")).launch()
```

02

基础组件之Pipeline

（1）什么是Pipeline

（2）Pipeline支持的任务类型

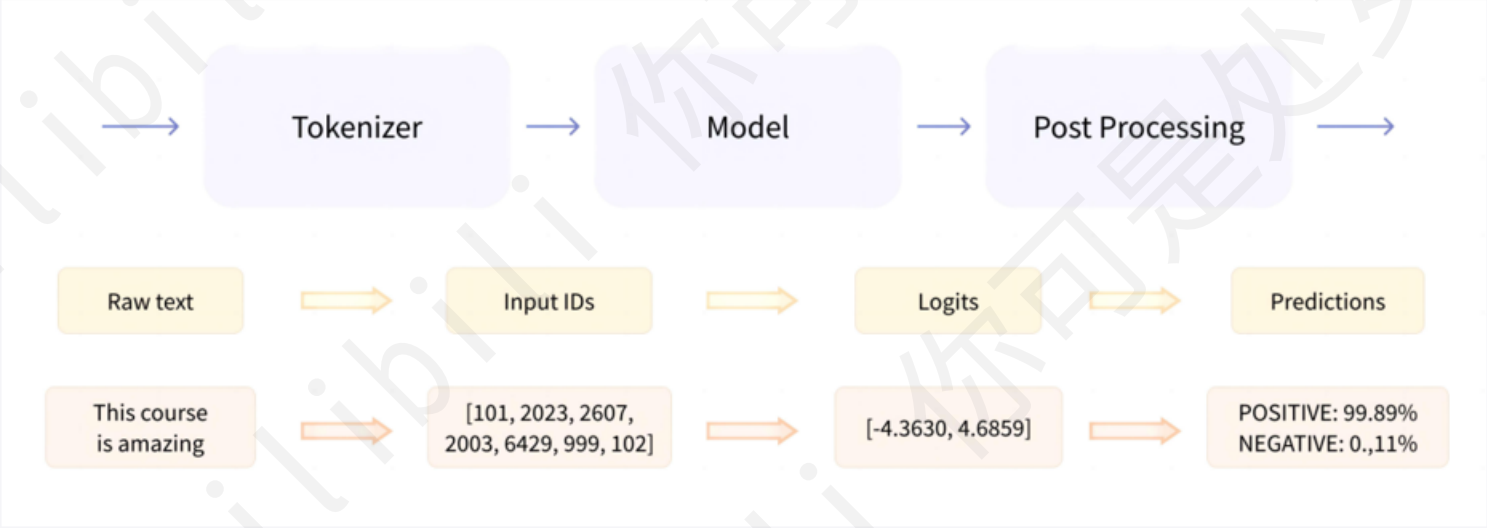
（3）Pipeline的创建与使用方式

（4）Pipeline的背后实现

基础组件之Pipeline

什么是Pipeline

- **Pipeline**
  - 将数据预处理、模型调用、结果后处理三部分组装成的流水线
  - 使我们能够直接输入文本便获得最终的答案



基础组件之Pipeline

Pipeline支持的任务类型

名称	任务类型
text-classification（sentiment-analysis）	text
token-classification（ner）	text
question-answering	text
fill-mask	text
summarization	text
translation	text
text2text-generation	text
text-generation	text
conversational	text
table-question-answering	text
zero-shot-classification	text
automatic-speech-recognition	multimodal
feature-extraction	multimodal
audio-classification	audio
visual-question-answering	multimodal
document-question-answering	multimodal
zero-shot-image-classification	multimodal
zero-shot-audio-classification	multimodal
image-classification	image
zero-shot-object-detection	multimodal
video-classification	video

基础组件之Pipeline

Pipeline创建与使用

- **根据任务类型直接创建Pipeline**
  - pipe = pipeline("text-classification")
- **指定任务类型，再指定模型，创建基于指定模型的Pipeline**
  - pipe = pipeline("text-classification", model="uer/roberta-base-finetuned-dianping-chinese")
- **预先加载模型，再创建Pipeline**
  - model = AutoModelForSequenceClassification.from_pretrained("uer/roberta-base-finetuned-dianping-chinese")
  - tokenizer = AutoTokenizer.from_pretrained("uer/roberta-base-finetuned-dianping-chinese")
  - pipe = pipeline("text-classification", model=model, tokenizer=tokenizer)
- **使用GPU进行推理加速**
  - pipe = pipeline("text-classification", model="uer/roberta-base-finetuned-dianping-chinese", device=0)

Pipeline的背后实现

- **Step1 初始化Tokenizer**
  - tokenizer = AutoTokenizer.from_pretrained("uer/roberta-base-finetuned-dianping-chinese")
- **Step2 初始化Model**
  - model = AutoModelForSequenceClassification.from_pretrained("uer/roberta-base-finetuned-dianping-chinese")
- **Step3 数据预处理**
  - input_text = "我觉得不太行！"
  - inputs = tokenizer(input_text, return_tensors="pt")
- **Step4 模型预测**
  - res = model(**inputs).logits
- **Step5 结果后处理**
  - pred = torch.argmax(torch.softmax(logits, dim=-1)).item()
  - result = model.config.id2label.get(pred)

03

基础组件之Tokenizer

（1）Tokenizer简介

（2）Tokenizer基本使用方法

（3）Fast / Slow Tokenizer

基础组件之Tokenizer

Tokenizer 简介

- **数据预处理**
  - **Step1 分词**：使用分词器对文本数据进行分词（字、字词）；
  - **Step2 构建词典**：根据数据集分词的结果，构建词典映射（这一步并不绝对，如果采用预训练词向量，词典映射要根据词向量文件进行处理）；
  - **Step3 数据转换**：根据构建好的词典，将分词处理后的数据做映射，将文本序列转换为数字序列；
  - **Step4 数据填充与截断**：在以batch输入到模型的方式中，需要对过短的数据进行填充，过长的数据进行截断，保证数据长度符合模型能接受的范围，同时batch内的数据维度大小一致。

现在：Tokenizer is all you need!

基础组件之Tokenizer

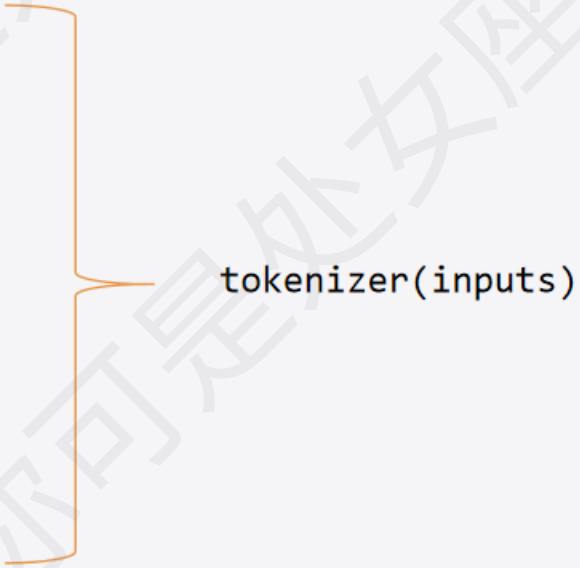
Tokenizer 基本使用

- 加载保存（from_pretrained / save_pretrained）
- 句子分词（tokenize）
- 查看词典（vocab）
- 索引转换（convert_tokens_to_ids / convert_ids_to_tokens）
- 填充截断（padding / truncation）
- 其他输入（attention_mask / token_type_ids）

基础组件之Tokenizer

Tokenizer 基本使用

- 加载保存（from_pretrained / save_pretrained）
- 句子分词（tokenize）
- 查看词典（vocab）
- 索引转换（convert_tokens_to_ids / convert_ids_to_tokens）
- 填充截断（padding / truncation）
- 其他输入（attention_mask / token_type_ids）



基础组件之Tokenizer

Fast / Slow Tokenizer

- **FastTokenizer**
  - 基于Rust实现，速度快
  - offsets_mapping、word_ids
- **SlowTokenizer**
  - 基于Python实现，速度慢

```
%%time
# 单条循环处理
for i in range(10000):
    fast_tokenizer(sen)
```

✓ 0.3s

```
CPU times: total: 78.1 ms
Wall time: 312 ms
```

```
%%time
# 处理batch数据
res = fast_tokenizer([sen] * 10000)
```

✓ 0.1s

```
CPU times: total: 359 ms
Wall time: 87.1 ms
```

```
%%time
# 单条循环处理
for i in range(10000):
    slow_tokenizer(sen)
```

✓ 0.8s

```
CPU times: total: 281 ms
Wall time: 872 ms
```

```
%%time
# 处理batch数据
res = slow_tokenizer([sen] * 10000)
```

✓ 0.7s

```
CPU times: total: 188 ms
Wall time: 705 ms
```

04 基础组件之Model

（1）Model 简介

（2）Model Head

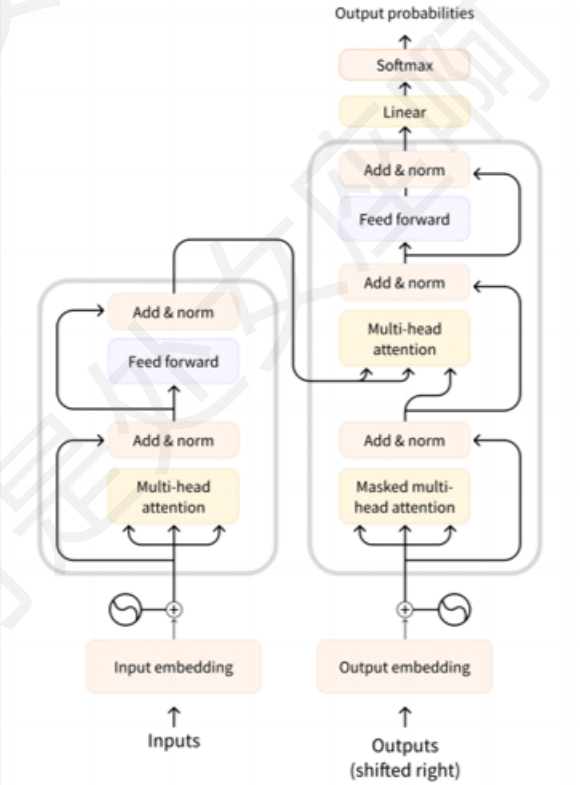
（3）Model 基本使用方法

（4）模型微调代码实例

基础组件之Model

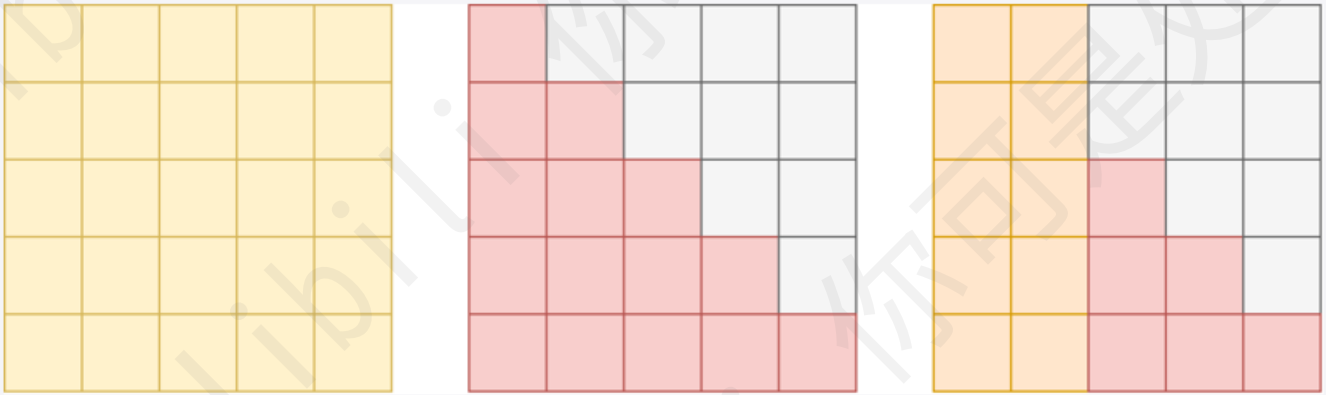
Model简介

- **Transformer**
  - 原始的Transformer为编码器（Encoder）、解码器（Decoder）模型
  - Encoder部分接收输入并构建其完整特征表示，Decoder部分使用Encoder的编码结果以及其他的输入生成目标序列
  - 无论是编码器还是解码器，均由多个TransformerBlock堆叠而成
  - TransformerBlock由注意力机制（Attention）和FFN组成
- **注意力机制**
  - 注意力机制的使用是Transformer的一个核心特性，在计算当前词的特征表示时，可以通过注意力机制有选择性的告诉模型要使用哪些上下文



Model简介

- **模型类型**
  - 编码器模型：自编码模型，使用Encoder，拥有双向的注意力机制，即计算每一个词的特征时都看到完整上下文
  - 解码器模型：自回归模型，使用Decoder，拥有单向的注意力机制，即计算每一个词的特征时都只能看到上文，无法看到下文
  - 编码器解码器模型：序列到序列模型，使用Encoder+Decoder，Encoder部分使用双向的注意力，Decoder部分使用单向注意力



编码器模型　　解码器模型　　编码器解码器模型

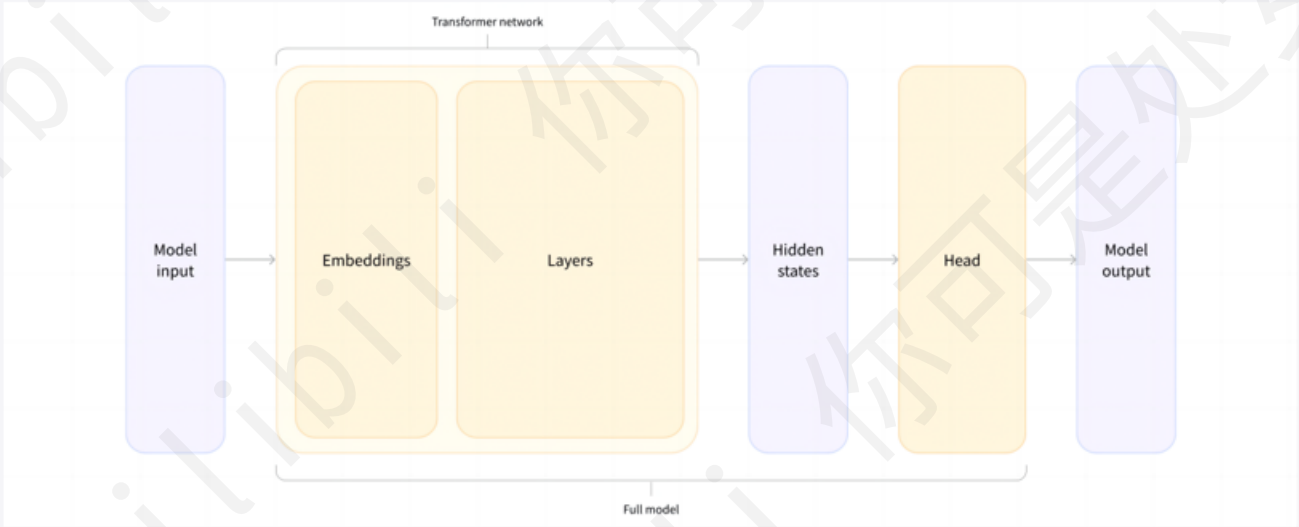
Model简介

- **模型类型**
  - 编码器模型：自编码模型，使用Encoder，拥有双向的注意力机制，即计算每一个词的特征时都看到完整上下文
  - 解码器模型：自回归模型，使用Decoder，拥有单向的注意力机制，即计算每一个词的特征时都只能看到上文，无法看到下文
  - 编码器解码器模型：序列到序列模型，使用Encoder+Decoder，Encoder部分使用双向的注意力，Decoder部分使用单向注意力

模型类型	常用预训练模型	适用任务
编码器模型，自编码模型	ALBERT, BERT, DistilBERT, RoBERTa	文本分类、命名实体识别、阅读理解
解码器模型，自回归模型	GPT, GPT-2, Bloom, LLaMA	文本生成
编码器解码器模型，序列到序列模型	BART, T5, Marian, mBART, GLM	文本摘要、机器翻译

Model Head

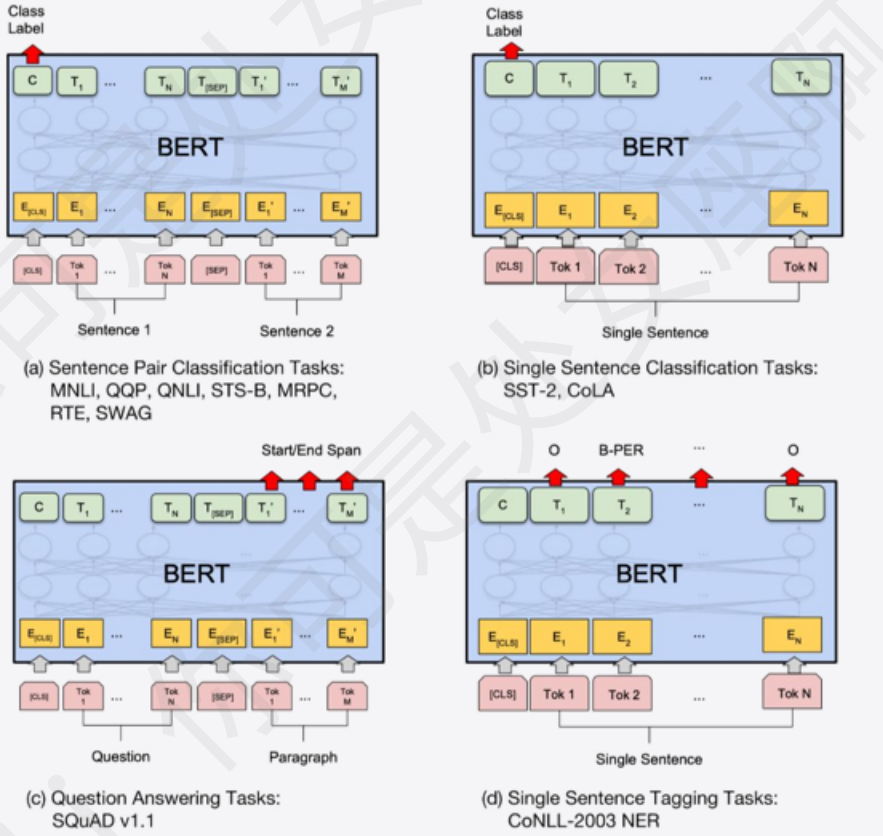
- **什么是Model Head**
  - Model Head 是连接在模型后的层，通常为1个或多个全连接层
  - Model Head 将模型的编码的表示结果进行映射，以解决不同类型的任务



基础组件之Model

Model Head

- **Transformers中的Model Head**
  - *Model（模型本身，只返回编码结果）
  - *ForCausalLM
  - *ForMaskedLM
  - *ForSeq2SeqLM
  - *ForMultipleChoice
  - *ForQuestionAnswering
  - *ForSequenceClassification
  - *ForTokenClassification
  - ... ...



Model基本使用方法

- **模型加载与保存**
  - 在线加载
  - 模型下载
  - 离线加载
  - 模型加载参数
- **模型调用**
  - 不带model head的模型调用
  - 带model head的模型调用

模型微调代码实例

- **任务类型**
  - 文本分类
- **使用模型**
  - hfl/rbt3
- **数据集地址**
  - https://github.com/SophonPlus/ChineseNlpCorpus

```
model.eval()
sen = "我觉得这家店的味道还是不错!"
with torch.inference_mode():
    inputs = tokenizer(sen, return_tensors="pt")
    batch = {k: v.cuda() for k, v in inputs.items()}
    logits = model(**batch).logits
    pred = torch.argmax(logits, dim=-1)
    print(f"评论: {sen}\n模型预测结果: {model.config.id2label.get(pred.item())}")
```

```
评论: 我觉得这家店的味道还是不错!
模型预测结果: 好评!
```

```
from transformers import pipeline

pipe = pipeline("text-classification", model=model, tokenizer=tokenizer, device=0)
```

```
pipe(sen)
```

```
[{'label': '好评! ', 'score': 0.9716703295707703}]
```

05

基础组件之Datasets

（1）Datasets简介

（2）Datasets基本使用

（3）Datasets加载本地数据集

（4）Datasets + DataCollator模型微调代码优化

基础组件之Datasets

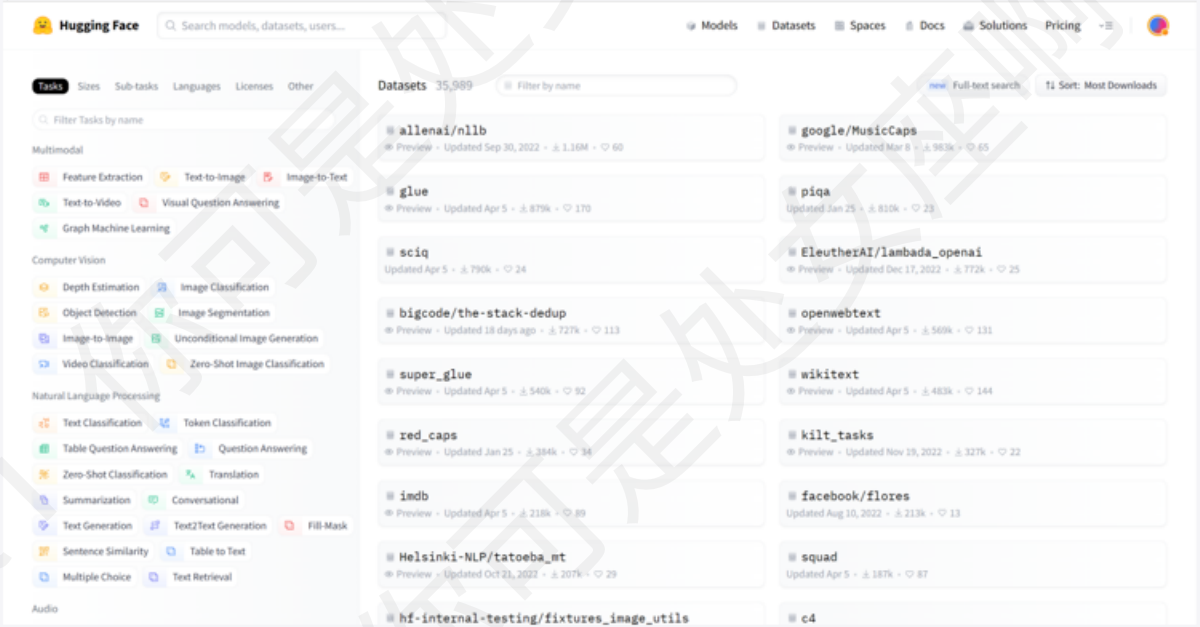
Datasets

- **简介**
  - datasets库是一个非常简单易用的数据集加载库，可以方便快捷的从本地或者HuggingFace Hub加载数据集
- **公开数据集地址**
  - https://huggingface.co/datasets
- **文档地址**
  - https://huggingface.co/docs/datasets/index

基础组件之Datasets

Datasets基本使用

- 加载在线数据集（load_dataset）
- 加载数据集某一项任务（load_dataset）
- 按照数据集划分进行加载（load_dataset）
- 查看数据集（index and slice）
- 数据集划分（train_test_split）
- 数据选取与过滤（select and filter）
- 数据映射（map）
- 保存与加载（save_to_disk / load_from_disk）



基础组件之Datasets

Datasets加载本地数据

- **直接加载文件作为数据集**
  - CSV、JSON
- **加载文件夹内全部文件作为数据集**
- **通过预先加载的其他格式转换加载数据集**
  - dict、pandas、list
- **通过自定义加载脚本加载数据集**
  - def _info(self)
  - def _split_generators(self, dl_manager)
  - def _generate_examples(self, filepath)

```
class CMRC2018TRIAL(datasets.GeneratorBasedBuilder):

    def _info(self) -> DatasetInfo:
        """
            info方法, 定义数据集的信息,这里要对数据的字段进行定义
        :return:
        """
        return datasets.DatasetInfo(…

    def _split_generators(self, dl_manager: DownloadManager):
        """
            返回datasets.SplitGenerator
            涉及两个参数: name和gen_kwargs
            name: 指定数据集的划分
            gen_kwargs: 指定要读取的文件的路径, 与_generate_examples的入参数一致
        :param dl_manager:
        :return: [ datasets.SplitGenerator ]
        """
        return [datasets.SplitGenerator(name=datasets.Split.TRAIN,
                                         gen_kwargs={"filepath": "./cmrc2018_trial.json"})]

    def _generate_examples(self, filepath):
        """
            生成具体的样本, 使用yield
            需要额外指定key, id从0开始自增就可以
        :param filepath:
        :return:
        """
        # Yields (key, example) tuples from the dataset
        with open(filepath, encoding="utf-8") as f:…
```

基础组件之Model

模型微调代码优化

- **任务类型**
  - 文本分类
- **使用模型**
  - hfl/rbt3
- **优化内容**
  - Datasets数据集加载
  - DataCollatorWithPadding

Step2 加载数据集

```
import torch
from datasets import load_dataset
from transformers import  DataCollatorWithPadding

tokenizer = AutoTokenizer.from_pretrained("hfl/rbt3")
```

```
dataset = load_dataset("csv", data_files="./04-datasets/ChnSentiCorp_htl_all.csv", split='train')
dataset = dataset.filter(lambda x: x["review"] is not None)
datasets = dataset.train_test_split(test_size=0.1)
datasets
```

输出已折叠 ...

Step3 数据预处理

```
def process_function(examples):
    tokenized_examples = tokenizer(examples["review"], max_length=128, truncation=True)
    tokenized_examples["labels"] = examples["label"]
    return tokenized_examples

tokenized_datasets = datasets.map(process_function, batched=True, remove_columns=dataset.column_names)
tokenized_datasets
```

06

基础组件之Evaluate

（1）Evaluate简介

（2）Evaluate基本使用

（3）Evaluate模型微调代码优化

基础组件之Evaluate

Evaluate

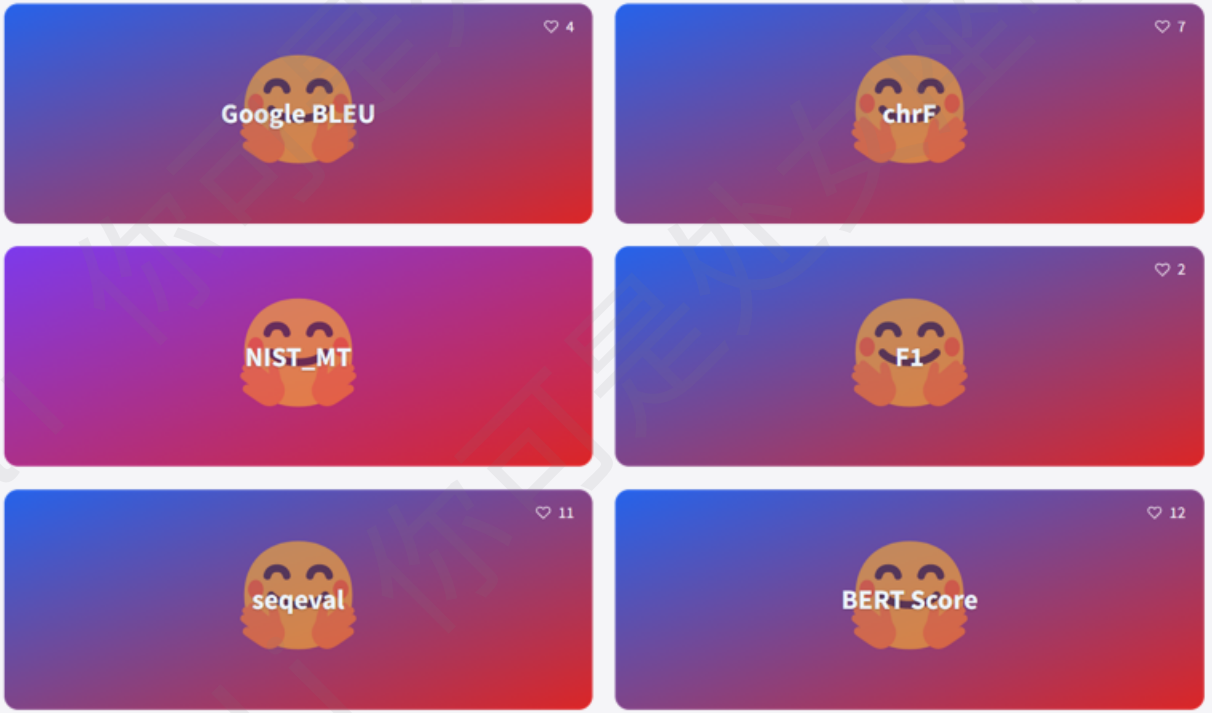
- **简介**
  - evaluate库是一个非常简单易用的机器学习模型评估函数库，

    只需要一行代码便可以加载各种任务的评估函数
- **函数库地址**
  - https://huggingface.co/evaluate-metric
- **文档地址**
  - https://huggingface.co/docs/evaluate/index

基础组件之Evaluate

Evaluate基本使用

- 查看支持的评估函数（list_evaluation_modules）
- 加载评估函数（load）
- 查看评估函数说明（inputs_description）
- 评估指标计算（compute）
  - 全局计算（compute）
  - 迭代计算（add / add_batch）
- 计算多个评估指标（combine）
- 评估结果对比可视化（radar_plot）



基础组件之Evaluate

模型微调代码优化

- **任务类型**
  - 文本分类
- **使用模型**
  - hfl/rbt3
- **优化内容**
  - Evaluate使用多个评估函数

Step7 训练与验证

```python
import evaluate

clf_metric = evaluate.combine(["accuracy", "f1", "recall", "precision"])

def evaluate():
    model.eval()
    with torch.inference_mode():
        for batch in validloader:
            if torch.cuda.is_available():
                batch = {k: v.cuda() for k, v in batch.items()}
            output = model(**batch)
            pred = torch.argmax(output.logits, dim=-1)
            clf_metric.add_batch(pred.long(), batch["labels"].long())
    return clf_metric.compute()

def train(epoch=3, log_step=100):
    global_step = 0
    for ep in range(epoch):
        model.train()
        for batch in trainloader:
            if torch.cuda.is_available():
                batch = {k: v.cuda() for k, v in batch.items()}
            optimizer.zero_grad()
            output = model(**batch)
            output.loss.backward()
            optimizer.step()
            if global_step % log_step == 0:
                print(f"ep: {ep}, global_step: {global_step}, loss: {output.loss.item()}")
            global_step += 1
        clf_result = evaluate()
        print(f"ep: {ep}, result: {clf_result}")
```

07

基础组件之Trainer

（1）Trainer简介

（2）TrainingArguments + Trainer代码优化

基础组件之Trainer

Trainer

- **简介**
  - Trainer是transformers库中提供的训练的函数，内部封装了完整的训练、评估逻辑，并集成了多种的后端，如DeepSpeed、Pytorch FSDP等，搭配TrainingArguments对训练过程中的各项参数进行配置，可以非常方便快捷地启动模型单机 / 分布式训练
  - 需要注意的是
    - 使用Trainer进行模型训练对模型的输入输出是有限制的，要求模型返回元组或者ModelOutput的子类
    - 如果输入中提供了labels，模型要能返回loss结果，如果是元组，要求loss为元组中第一个值
- **文档地址**
  - https://huggingface.co/docs/transformers/main_classes/trainer#trainer

基础组件之Trainer

模型微调代码优化

- **任务类型**
  - 文本分类
- **使用模型**
  - hfl/rbt3
- **优化内容**
  - 使用Trainer + TrainingArgument优化训练流程

完整使用流程

```
from transformers import Trainer, TrainingArguments
# 创建TrainingArguments
training_args = TrainingArguments(...)
# 创建Trainer
trainer = Trainer(args=training_args, ...)
# 模型训练
trainer.train()
# 模型评估
trainer.evaluate()
# 模型预测
trainer.predict()
```

基础入门篇
完结

实战演练篇

基于Transformers的NLP解决方案

你可是处女座啊

目录

目录

01

基于Transformers的NLP解决方案

（1）基础组件知识回顾

（2）基于Transformers的NLP解决方案

（3）显存优化策略，4G显存跑BERT-Large

基础组件知识回顾

截止目前讲的基础组件

- **Pipeline**
  - 流水线，用于模型推理，封装了完整的推理逻辑，包括数据预处理、模型预测及后处理
- **Tokenizer**
  - 分词器，用于数据预处理，将原始文本输入转换为模型的输入，包括input_ids、attention_mask等
- **Model**
  - 模型，用于加载、创建、保存模型，对Pytorch中的模型进行了封装，同时更好的支持预训练模型
- **Datasets**
  - 数据集，用于数据集加载与预处理，支持加载在线与本地的数据集，提供了数据集层面的处理方法
- **Evaluate**
  - 评估函数，用于对模型的结果进行评估，支持多种任务的评估函数
- **Trainer**
  - 训练器，用于模型训练、评估，支持丰富的配置选项，快速启动模型训练流程

基于Transformers的NLP解决方案

- **以文本分类为例**
  - Step1 导入相关包 > General
  - Step2 加载数据集 > Datasets
  - Step3 数据集划分 > Datastes
  - Step4 数据集预处理 > Tokenizer + Datasets
  - Step5 创建模型 > Model
  - Step6 设置评估函数 > Evaluate
  - Step7 配置训练参数 > TrainingArguments
  - Step8 创建训练器 > Trainer + Data Collator
  - Step9 模型训练、评估、预测（数据集） > Trainer
  - Step10 模型预测（单条） > Pipeline

Transformers显存优化

显存优化策略，4G显存也能跑BERT-Large

- **显存占用简单分析**
  - 模型权重
    - 4Bytes * 模型参数量
  - 优化器状态
    - 8Bytes * 模型参数量，对于常用的AdamW优化器而言
  - 梯度
    - 4Bytes * 模型参数量
  - 前向激活值
    - 取决于序列长度、隐层维度、Batch大小等多个因素

Transformers显存优化

显存优化策略，4G显存也能跑BERT-Large

- **显存优化策略**
  - hfl/chinese-macbert-large，330M

优化策略	优化对象	显存占用	训练时间
Baseline (BS 32, MaxLength 128)	-	15.2G	64s
+ Gradient Accumulation (BS 1, GA 32)	前向激活值	7.4G	259s
+ Gradient Checkpoints (BS 1, GA 32)	前向激活值	7.2G	422s
+ Adafactor Optiomizer (BS 1, GA 32)	优化器状态	5.0G	406s
+ Freeze Model (BS 1, GA 32)	前向激活值 / 梯度	3.5G	178s
+ Data Length (BS 1, GA 32, MaxLength 32)	前向激活值	3.4G	126s

关于参数高效微调（如Lora）、cpu offload、flash attention等技巧将在LLM章节进行讲解

02

实战演练之命名实体识别

（1）命名实体识别任务介绍

（2）基于Transfromers的解决方案

（3）代码实战演练

命名实体识别任务介绍

命名实体识别简介

- **什么是命名实体识别任务**
  - 命名实体识别（Named Entity Recognition，简称NER）是指识别文本中具有特定意义的实体，主要包括人名、地名、机构名、专有名词等。通常包括两部分：（1）实体边界识别；（2）确定实体类别（人名、地名、机构名或其他）。
  - 例：

“小明在北京上班”

实体类别	实体
地点	北京
人物	小明

命名实体识别任务介绍

命名实体识别简介

- **数据标注体系**
  - IOB1、IOB2、IOE1、IOE2、IOBES、BILOU
- **IOB2标注**
  - I表示实体内部，O表示实体外部，B表示实体开始
  - B/I-XXX，XXX表示具体的类别

标记	说明
B-Person	人名开始
I- Person	人名中间
B-Organization	组织名开始
I-Organization	组织名中间
O	非命名实体

- **IOBES标注**
  - I表示实体内部，O表示实体外部，B表示实体开始，E表示实体结束，S表示一个词单独形成一个命名实体
  - 有时也会使用M代替I，但本质是同一含义

命名实体识别任务介绍

命名实体识别简介

- **评估指标**
  - Precision、Recall、F1
- **简单示例**

Sen:	The	Hospital	said	it	would	probably	know	by	Tuesday	whether	its	patients
Gold:	**b-AGENT**	**e-AGENT**	o	o	**b-DSE**	**m-DSE**	**e-DSE**	o	o	**b-TARGET**	**m-TARGET**	**m-TARGET**
Predict:	o	e-AGENT	b-DSE	o	b-DSE	m-DSE	e-DSE	o	o	b-AGENT	b-DSE	b-TARGET

Sen:	had	Congo	Fever	.
Gold:	**m-TARGET**	**m-TARGET**	**e-TARGET**	o
Predict:	m-TARGET	e-TARGET	o	o

predict_num=2
gold_num=3
correct_num=1

$P=\frac{1}{2}$ $R=\frac{1}{3}$ $F1=\frac{2*\frac{1}{2}*\frac{1}{3}}{\frac{1}{2}+\frac{1}{3}}$

基于Transformers的解决方案

- **模型结构**
  - *ModelForTokenClassification

```python
class BertForTokenClassification(BertPreTrainedModel):
    _keys_to_ignore_on_load_unexpected = [r"pooler"]

    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        self.bert = BertModel(config, add_pooling_layer=False)
        classifier_dropout = (
            config.classifier_dropout if config.classifier_dropout is not None else config.hidden_dropout_prob
        )
        self.dropout = nn.Dropout(classifier_dropout)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

        # Initialize weights and apply final processing
        self.post_init()
```

基于Transformers的解决方案

- **模型结构**
  - *ModelForTokenClassification

```
outputs = self.bert(
    input_ids,
    attention_mask=attention_mask,
    token_type_ids=token_type_ids,
    position_ids=position_ids,
    head_mask=head_mask,
    inputs_embeds=inputs_embeds,
    output_attentions=output_attentions,
    output_hidden_states=output_hidden_states,
    return_dict=return_dict,
)

sequence_output = outputs[0]

sequence_output = self.dropout(sequence_output)
logits = self.classifier(sequence_output)

loss = None
if labels is not None:
    loss_fct = CrossEntropyLoss()
    loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
```

基于Transformers的解决方案

- **评估函数**
  - seqeval
  - 需要额外安装 seqeval
    - pip install seqeval
    - 安装过程中报错：Microsoft Visual C++ 14.0 or greater is required. Get it with "Microsoft C++ Build Tools
    - 进入https://my.visualstudio.com，下载C++ build tools，安装
  - evaluate.load（“seqeval”）

代码实战演练

- **数据集**
  - peoples_daily_ner
- **预训练模型**
  - hfl/chinese-macbert-base

03

实战演练之机器阅读理解

（1）机器阅读理解任务介绍

（2）基于Transfromers的解决方案

（3）代码实战演练（上）

（4）代码实战演练（下）

机器阅读理解任务介绍

机器阅读理解简介

- **什么是机器阅读理解任务**
  - 机器阅读理解（Machine Reading Comprehension，简称MRC）是一项通过让机器回答基于给定上下文的问题来测试机器理解自然语言的程度的任务，简单来说即**给定一个或者多个文档P，以及一个问题Q，输出问题Q的答案A。**
  - 机器阅读理解任务的形式是较为多样化的，常见的类型包括完型填空式、答案选择式的、片段抽取式的、自由生成式，本次课程讲解的内容为**片段抽取式的机器阅读理解，即问题Q的答案A在文档P中，A是P中的一个连续片段。**

机器阅读理解任务介绍

机器阅读理解简介

- **机器阅读理解任务样例**

工商协进会报告，12月消费者信心上升到78.1，明显高于11月的72。另据《华尔街日报》报道，2013年是1995年以来美国股市表现最好的一年。这一年里，投资美国股市的明智做法是追着“傻钱”跑。所谓的“傻钱”策略，其实就是买入并持有美国股票这样的普通组合。这个策略要比对冲基金和其它专业投资者使用的更为复杂的投资方法效果好得多。

Q1：消费者信心指数由什么机构发布?

A1：工商协进会

Q2：12月的消费者信心指数是多少?

A2： 78.1

Q3：什么是傻钱策略?

A3： 买入并持有美国股票这样的普通组合

如何为机器阅读理解任务建模？

机器阅读理解任务介绍

机器阅读理解简介

- **机器阅读理解任务样例**

工商协进会报告，12月消费者信心上升到78.1，明显高于11月的72。另据《华尔街日报》报道，2013年是1995年以来美国股市表现最好的一年。这一年里，投资美国股市的明智做法是追着“傻钱”跑。所谓的“傻钱”策略，其实就是买入并持有美国股票这样的普通组合。这个策略要比对冲基金和其它专业投资者使用的更为复杂的投资方法效果好得多。

Q1：消费者信心指数由什么机构发布?

A1：工商协进会

Q2：12月的消费者信心指数是多少?

A2： 78.1

Q3：什么是傻钱策略?

A3： 买入并持有美国股票这样的普通组合

如何为机器阅读理解任务建模？

定位答案在文档中的起始位置和结束位置

机器阅读理解任务介绍

机器阅读理解简介

- **机器阅读理解数据集格式**
  - CMRC2018

```
{
    "context": "《战国无双3》是由光荣和ω-force开发的战国无双系列的正统第三续作。本作以三大故事为主轴，分别是以武田信玄等人为主的《关东三国志》，织田信长等人为主的《战国三杰》，石田三成等人为主的《关原的年轻武者》，丰富游戏内的剧情。此部份专门介绍角色，欲知武...",
    "id": "DEV_0_QUERY_0",
    "question": "《战国无双3》是由哪两个公司合作开发的？",
    "answers": {
            "answer_start": [11, 11],
            "text": ["光荣和ω-force", "光荣和ω-force"]
     },
}
```

机器阅读理解任务介绍

机器阅读理解简介

- **评估指标**
  - 精准匹配度（Exact Match，EM）：计算预测结果与标准答案是否完全匹配。
  - 模糊匹配度（F1）：计算预测结果与标准答案之间字级别的匹配程度。
- **简单示例**
  - 数据
    - 模型预测结果：北京
    - 真实标签结果：北京天安门
  - 计算结果
    - EM = 0，P = 2/2 R=2/5 F1 = (2 * 2/2 * 2/5) / (2/2 + 2/5) = 4/7 ~=0.57

基于Transformers的解决方案

- **数据预处理**
  - 数据处理格式

  - 如何准确定位答案位置
    - start_positions / end_positions
    - offset_mapping
  - Context过长如何解决
    - 策略1 直接截断，简单易实现，但是会损失答案靠后的数据，因为无法定位答案
    - 策略2 滑动窗口，实现较为复杂，会丢失部分上下文，但是综合来看损失较小

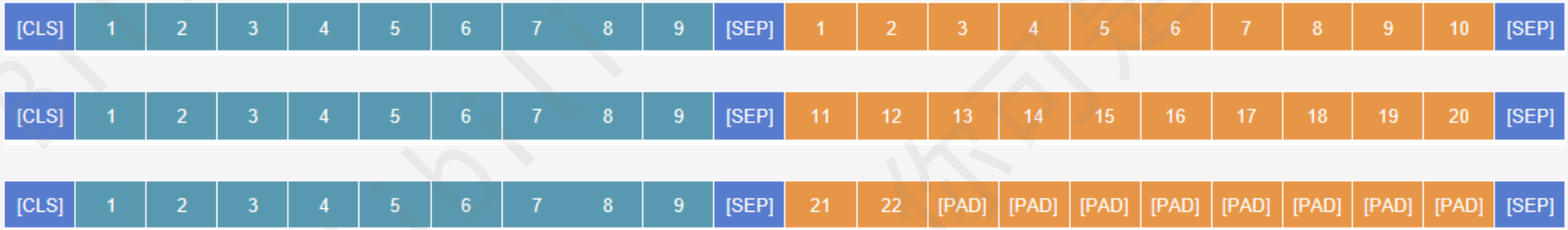
代码实战演练

基于滑动窗口的数据处理

- **滑动窗口**
  - Context

  - Query

  - Input



代码实战演练

基于滑动窗口的数据处理

- **带有overflow的滑动窗口**
  - Context

  - Query

  - Input

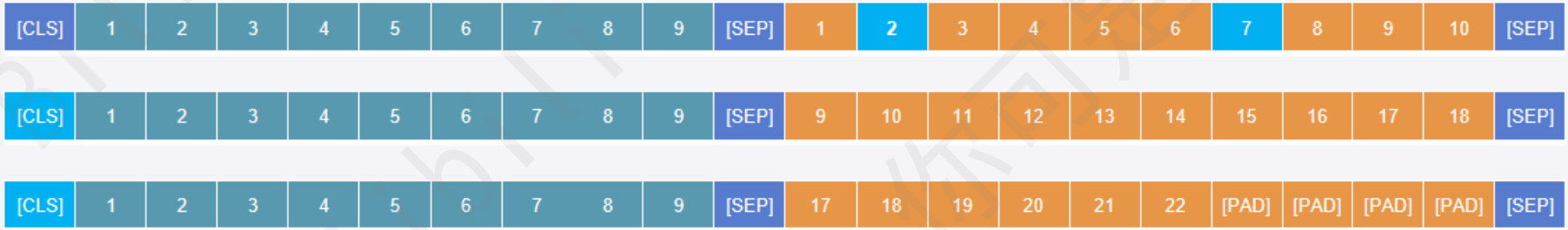
代码实战演练

基于滑动窗口的结果预测

- **结果预测**
  - Context

  - Query

  - Input



基于Transformers的解决方案

- **模型结构**
  - *ModelForQuestionAnswering

```python
class BertForQuestionAnswering(BertPreTrainedModel):
    _keys_to_ignore_on_load_unexpected = [r"pooler"]

    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        self.bert = BertModel(config, add_pooling_layer=False)
        self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels)

        # Initialize weights and apply final processing
        self.post_init()
```

基于Transformers的解决方案

- **模型结构**
  - *ModelForQuestionAnswering

```
outputs = self.bert(
    input_ids,
    attention_mask=attention_mask,
    token_type_ids=token_type_ids,
    position_ids=position_ids,
    head_mask=head_mask,
    inputs_embeds=inputs_embeds,
    output_attentions=output_attentions,
    output_hidden_states=output_hidden_states,
    return_dict=return_dict,
)

sequence_output = outputs[0]

logits = self.qa_outputs(sequence_output)   # [bs, seq_len, 2]
start_logits, end_logits = logits.split(1, dim=-1)  # [bs, seq_len, 1], [bs, seq_len, 1]
start_logits = start_logits.squeeze(-1).contiguous()     # [bs, seq_len]
end_logits = end_logits.squeeze(-1).contiguous()         # [bs, seq_len]
```

基于Transformers的解决方案

- **模型结构**
  - *ModelForQuestionAnswering

```
total_loss = None
if start_positions is not None and end_positions is not None:
    # If we are on multi-GPU, split add a dimension
    if len(start_positions.size()) > 1:
        start_positions = start_positions.squeeze(-1)
    if len(end_positions.size()) > 1:
        end_positions = end_positions.squeeze(-1)
    # sometimes the start/end positions are outside our model inputs, we ignore these terms
    ignored_index = start_logits.size(1)
    start_positions = start_positions.clamp(0, ignored_index)
    end_positions = end_positions.clamp(0, ignored_index)

    loss_fct = CrossEntropyLoss(ignore_index=ignored_index)
    start_loss = loss_fct(start_logits, start_positions)
    end_loss = loss_fct(end_logits, end_positions)
    total_loss = (start_loss + end_loss) / 2
```

代码实战演练（上，截断策略版本）

- **数据集**
  - cmrc2018
- **预训练模型**
  - hfl/chinese-macbert-base
- **数据集处理方式**
  - 对context进行截断处理

代码实战演练（下，滑动窗口版本）

- **数据集**
  - cmrc2018
- **预训练模型**
  - hfl/chinese-macbert-base
- **数据集处理方式**
  - 对context进行滑动窗口处理
- **相关包安装**
  - `pip install nltk`
  - `nltk.download(“punkt”)`

04

实战演练之多项选择

（1）多项选择任务介绍

（2）基于Transfromers的解决方案

（3）代码实战演练

多项选择任务介绍

多项选择简介

- **什么是多项选择任务**
  - 多项选择（Multiple Choice）任务是机器阅读理解任务中的一种形式，相较之前的机器阅读理解定义而言，多项选择任务还需要提供答案的候选项，即**给定一个或者多个文档P，以及一个问题Q和对应的多个答案候选，输出问题Q的答案A，A是答案候选中的某一个选项。**
  - 当然，更广泛的来看，多项选择也不局限于一定是阅读理解，也可以理解为是一种匹配，关于文本匹配的概念，将在下次课程进行介绍

多项选择任务介绍

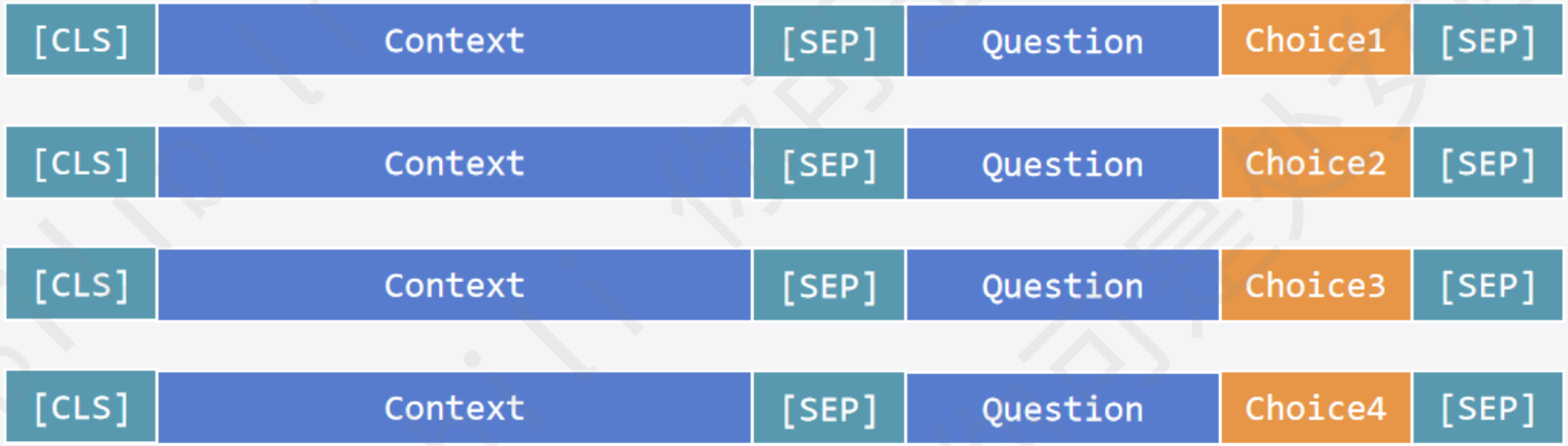
多项选择简介

- **多项选择任务示例**
  - Context：老师把一个大玻璃瓶子带到学校，瓶子里装着满满的石头、玻璃碎片和沙子。之后，老师请学生把瓶子里的东西都倒出来，然后再装进去，先从沙子开始。每个学生都试了试，最后都发现没有足够的空间装所有的石头。老师指导学生重新装这个瓶子。这次，先从石头开始，最后再装沙子。石头装进去后，沙子就沉积在石头的周围，最后，所有东西都装进瓶子里了。老师说：“如果我们先从小的东西开始，把小东西装进去之后，大的石头就放不进去了。生活也是如此，如果你的生活先被不重要的事挤满了，那你就无法再装进更大、更重要的事了。”
  - Question：正确的装法是，先装?
  - Choices / Candidates：["小东西", "大东西", "轻的东西", "软的东西"]
  - Answer：大东西

本质上就是一项分类任务

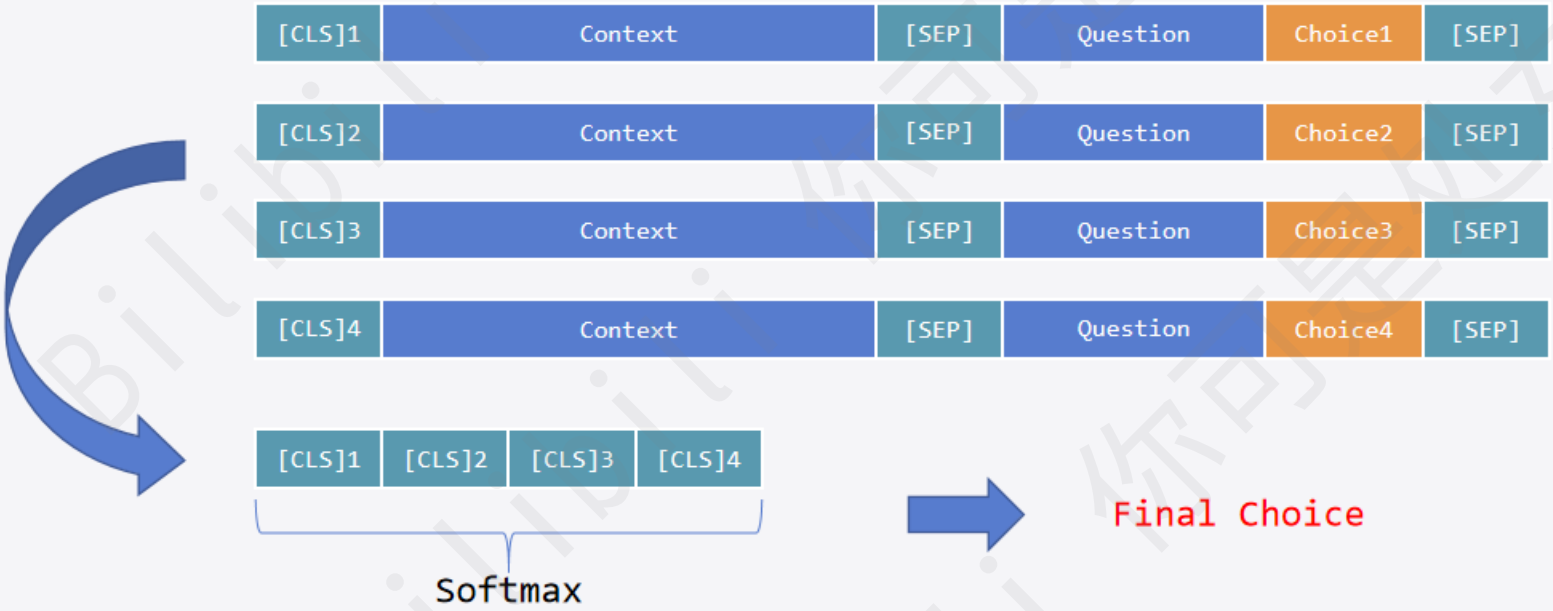
基于Transformers的解决方案

- **数据预处理**
  - 数据处理格式



基于Transformers的解决方案

- **训练与推理**
  - 基本原理



基于Transformers的解决方案

- **模型结构**
  - *ModelForMultipleChoice

```python
class BertForMultipleChoice(BertPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)

        self.bert = BertModel(config)
        classifier_dropout = (
            config.classifier_dropout if config.classifier_dropout is not None else config.hidden_dropout_prob
        )
        self.dropout = nn.Dropout(classifier_dropout)
        self.classifier = nn.Linear(config.hidden_size, 1)

        # Initialize weights and apply final processing
        self.post_init()
```

基于Transformers的解决方案

- **模型结构**
  - *ModelForMultipleChoice

```
num_choices = input_ids.shape[1] if input_ids is not None else inputs_embeds.shape[1]

# [batch * num_choices, seq_len]
input_ids = input_ids.view(-1, input_ids.size(-1)) if input_ids is not None else None
attention_mask = attention_mask.view(-1, attention_mask.size(-1)) if attention_mask is not None else None
token_type_ids = token_type_ids.view(-1, token_type_ids.size(-1)) if token_type_ids is not None else None
position_ids = position_ids.view(-1, position_ids.size(-1)) if position_ids is not None else None
inputs_embeds = (
    inputs_embeds.view(-1, inputs_embeds.size(-2), inputs_embeds.size(-1))
    if inputs_embeds is not None
    else None
)
```

基于Transformers的解决方案

- **模型结构**
  - *ModelForMultipleChoice

```
outputs = self.bert(
    input_ids,
    attention_mask=attention_mask,
    token_type_ids=token_type_ids,
    position_ids=position_ids,
    head_mask=head_mask,
    inputs_embeds=inputs_embeds,
    output_attentions=output_attentions,
    output_hidden_states=output_hidden_states,
    return_dict=return_dict,
)

pooled_output = outputs[1]  # [batch * num_choices, hidden_dim]

pooled_output = self.dropout(pooled_output) # [batch * num_choices, hidden_dim]
logits = self.classifier(pooled_output)     # [batch * num_choices, 1]
reshaped_logits = logits.view(-1, num_choices)  # [batch, num_choices]
```

代码实战演练

- **数据集**
  - C3
- **预训练模型**
  - hfl/chinese-macbert-base

05

实战演练之文本相似度

（1）文本匹配与文本相似度介绍

（2）基于Transfromers的解决方案

（3）代码实战演练（上，交互/单塔模型）

（4）代码实战演练（下，匹配/双塔模型）

抽奖活动

- **活动奖品**
  - AutoDL 充值 50 元
  - 抽 2 人
- **参与方式**
  - 本期视频 点赞+评论 即可参与抽奖
- **截止时间**
  - 2023.07.21

后续将作为长期的一个活动

创作激励每达500，取20%抽奖回馈给大家

希望各位小伙伴多多三连支持咯！

文本匹配与文本相似度简介

- **什么是文本匹配任务**
  - 文本匹配（Text Match）是一个较为宽泛的概念，基本上**只要涉及到两段文本之间关系的，都可以被看作是一种文本匹配的任务，只是在具体的场景下，不同的任务对匹配二字的定义可能是存在差异**的，具体的任务场景包括文本相似度计算、问答匹配、对话匹配、文本推理等等，另外，如之前介绍的抽取式机器阅读理解和多项选择，本质上也都是文本匹配。
  - 本次课程重点关注文本相似度任务，即判断两段文本是不是表达了同样的语义

文本匹配与文本相似度任务介绍

文本匹配与文本相似度简介

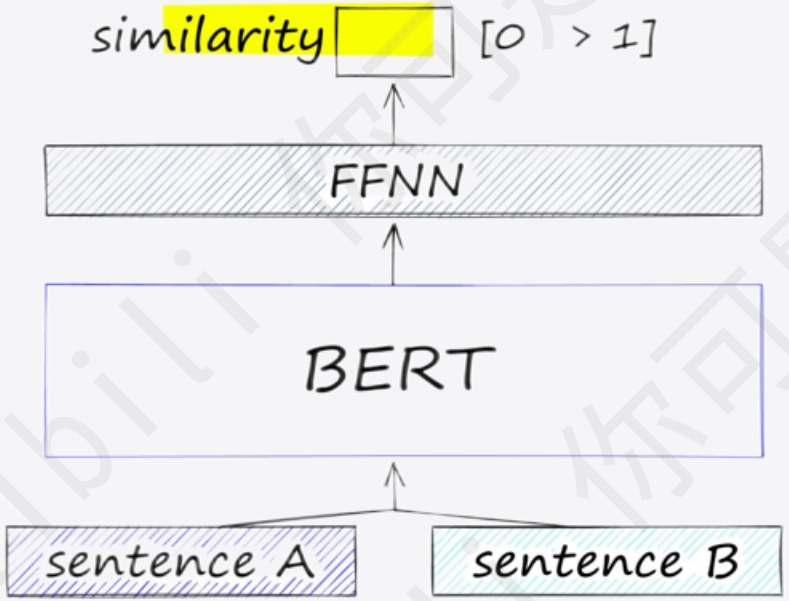
- **文本相似度**

Sentence A	Sentence B	Label
找一部小时候的动画片	求一部小时候的动画片。谢了	1
别急呀，我的朋友。	你一定要看我一下。	0
明天多少度啊	明天气温多少度啊	1
可怕的事情终于发生。	你到底想说什么?	0

本质上是一项分类任务

基于Transformers的解决方案

- **最直观的解决方案**
  - 交互策略，输入句子对，对是否相似进行学习



基于Transformers的解决方案

- **最直观的解决方案**
  - 数据处理格式

  - 模型训练方式

代码实战演练（交互模式/单塔）

- **数据集**
  - simCLUE / train_pair_1w.json
  - https://github.com/CLUEbenchmark/SimCLUE/tree/main
- **预训练模型**
  - hfl/chinese-macbert-base

基于Transformers的解决方案

- **最直观的解决方案**
  - 数据处理格式

  - 模型训练方式

如何从多个候选文本中找出最相似的？

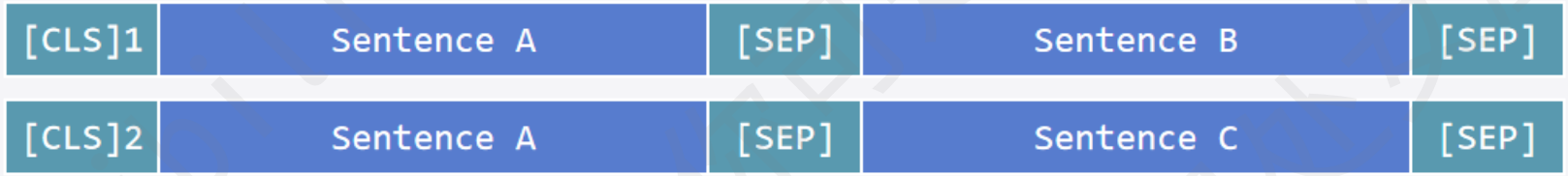
基于Transformers的解决方案

- **最直观的解决方案**
  - 数据处理格式

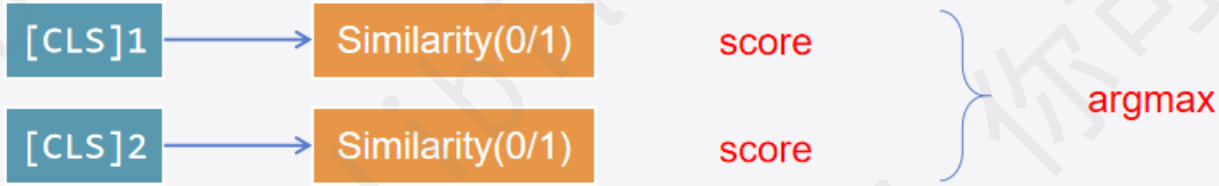
  - 模型训练方式

基于Transformers的解决方案

- **最直观的解决方案**
  - 数据处理格式



  - 模型训练方式



代码实战演练（交互模式/单塔）

- **数据集**
  - simCLUE / train_pair_1w.json
  - https://github.com/CLUEbenchmark/SimCLUE/tree/main
- **预训练模型**
  - hfl/chinese-macbert-base

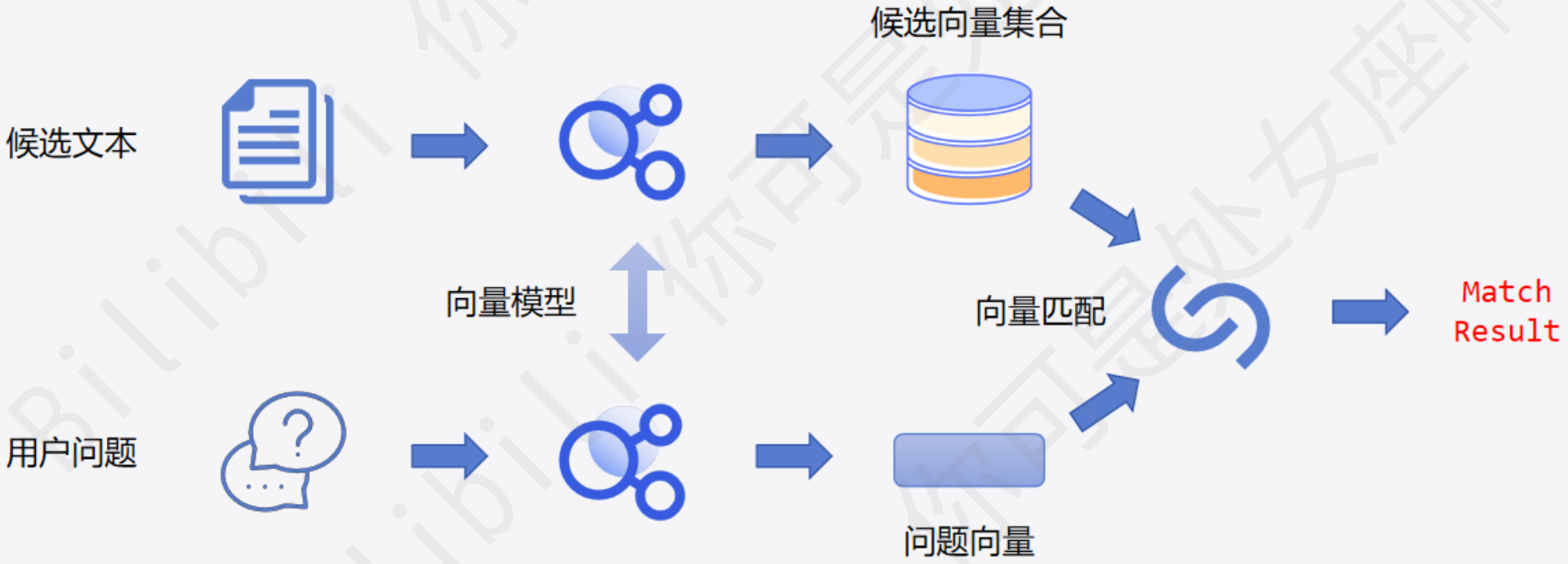
基于Transformers的解决方案

- **再看基于交互策略解决方案**
  - 效率问题
    - 1个待匹配文本， 1 000 000候选文本
    - 假设1次推理匹配 20ms
    - 1 000 000候选文本则需推理匹配1 000 000次
    - 消耗时间：20 000 000ms = 20 000s ~= 333.33min ~= 5.56h

效率极低，每次都需要与全量数据进行模型推理，数据量较大时很难满足时延要求，如何解决？

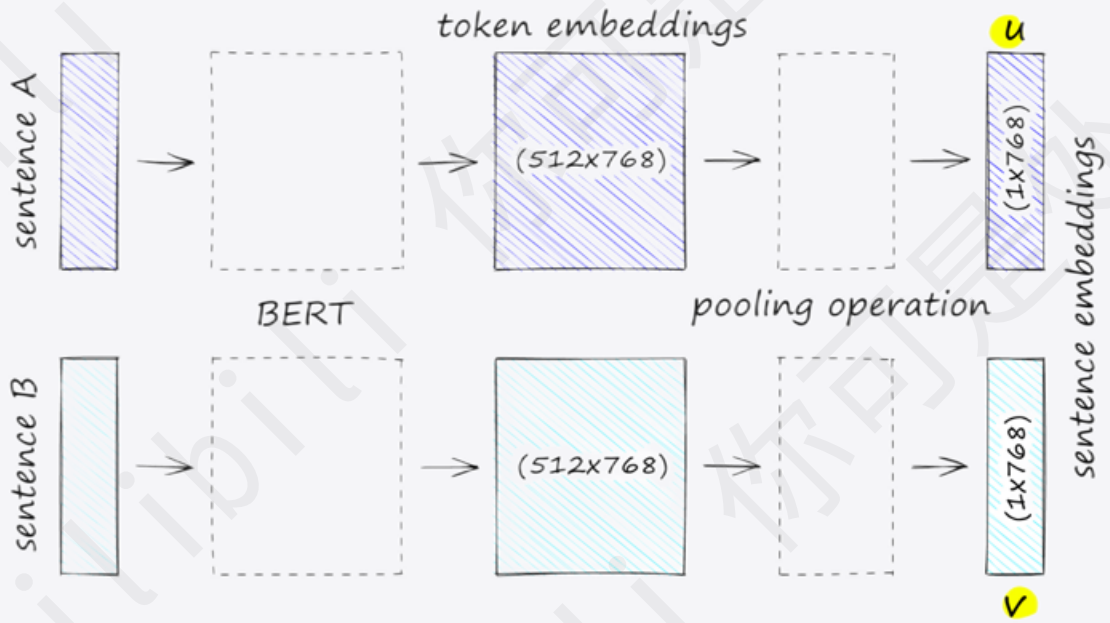
基于Transformers的解决方案

- 基于向量匹配的解决方案



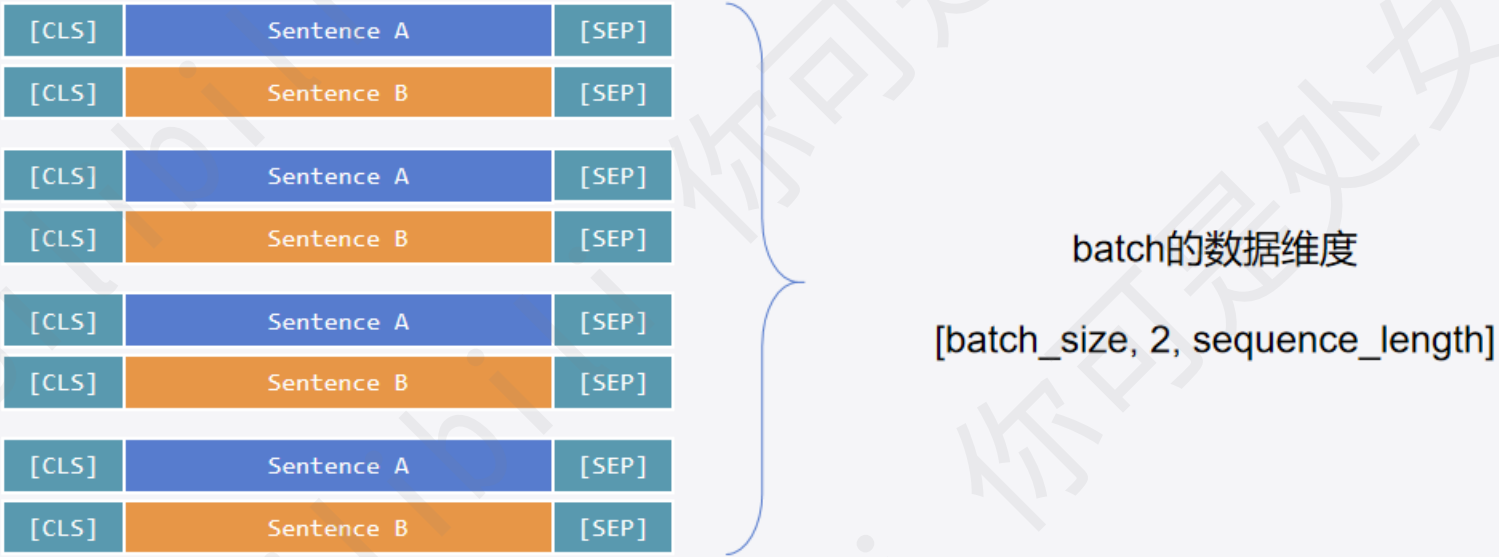
基于Transformers的解决方案

- **基于向量匹配的解决方案**
  - 向量匹配训练，分别对句子进行编码，目标是让两个相似句子的相似度分数尽可能接近1



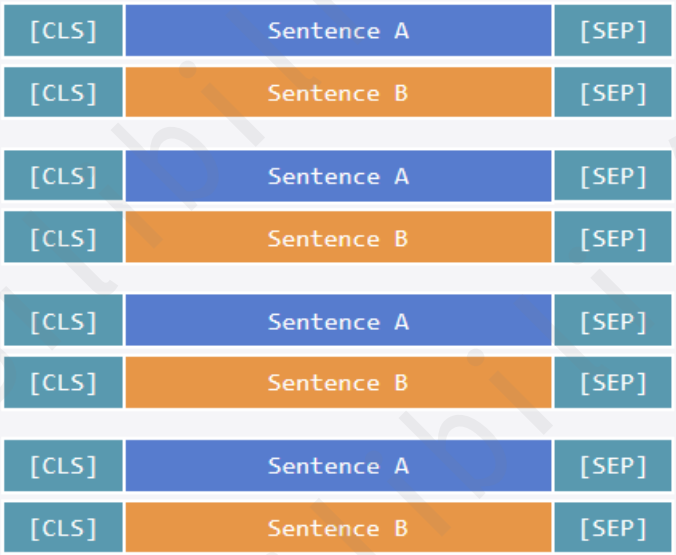
基于Transformers的解决方案

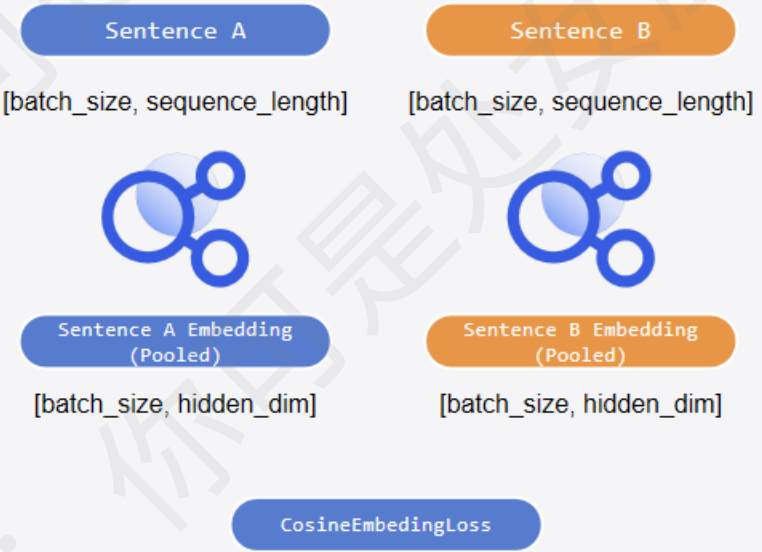
- **数据预处理**
  - 数据处理格式，与多项选择类似，因为要保证每个batch内都是对的数据



基于Transformers的解决方案

- **模型结构**
  - 没有预定义的模型，需要自行实现





基于Transformers的解决方案

- **模型结构**
  - CosineEmbeddingLoss

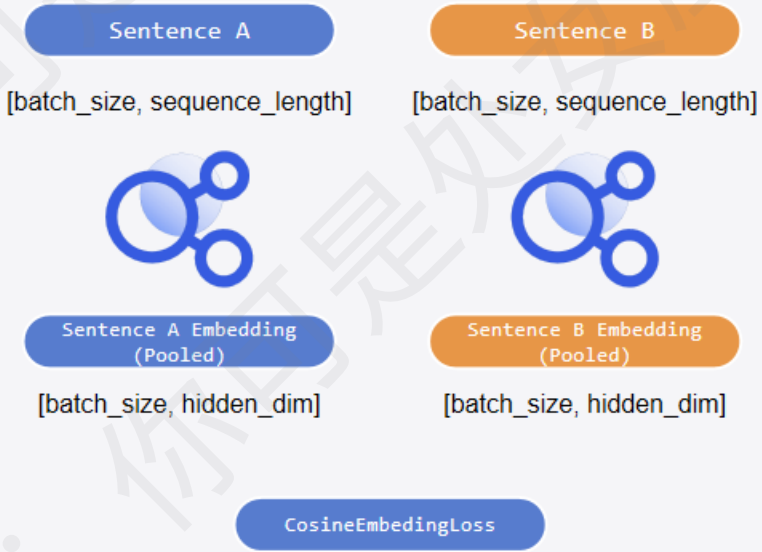
COSINEEMBEDDINGLOSS

CLASS torch.nn.CosineEmbeddingLoss(*margin=0.0*, *size_average=None*, *reduce=None*, *reduction='mean'*) [SOURCE]

Creates a criterion that measures the loss given input tensors x_1, x_2 and a *Tensor* label y with values 1 or -1. This is used for measuring whether two inputs are similar or dissimilar, using the cosine similarity, and is typically used for learning nonlinear embeddings or semi-supervised learning.

The loss function for each sample is:

$$\text{loss}(x, y) = \begin{cases} 1 - \cos(x_1, x_2), & \text{if } y = 1 \\ \max(0, \cos(x_1, x_2) - \text{margin}), & \text{if } y = -1 \end{cases}$$



代码实战演练（匹配模式/双塔）

- **数据集**
  - simCLUE / train_pair_1w.json
  - https://github.com/CLUEbenchmark/SimCLUE/tree/main
- **预训练模型**
  - hfl/chinese-macbert-base

代码实战演练

小结

- **文本相似度模型**
  - 基于交互策略的单塔模型
  - 基于向量匹配的双塔模型
- **更加便捷有效的工具**
  - sentence-transformers https://www.sbert.net/
  - text2vec https://github.com/shibing624/text2vec
  - uniem https://github.com/wangyuxinwhy/uniem

06

实战演练之
检索式对话机器人

（1）对话机器人介绍

（2）解决方案与代码实战演练

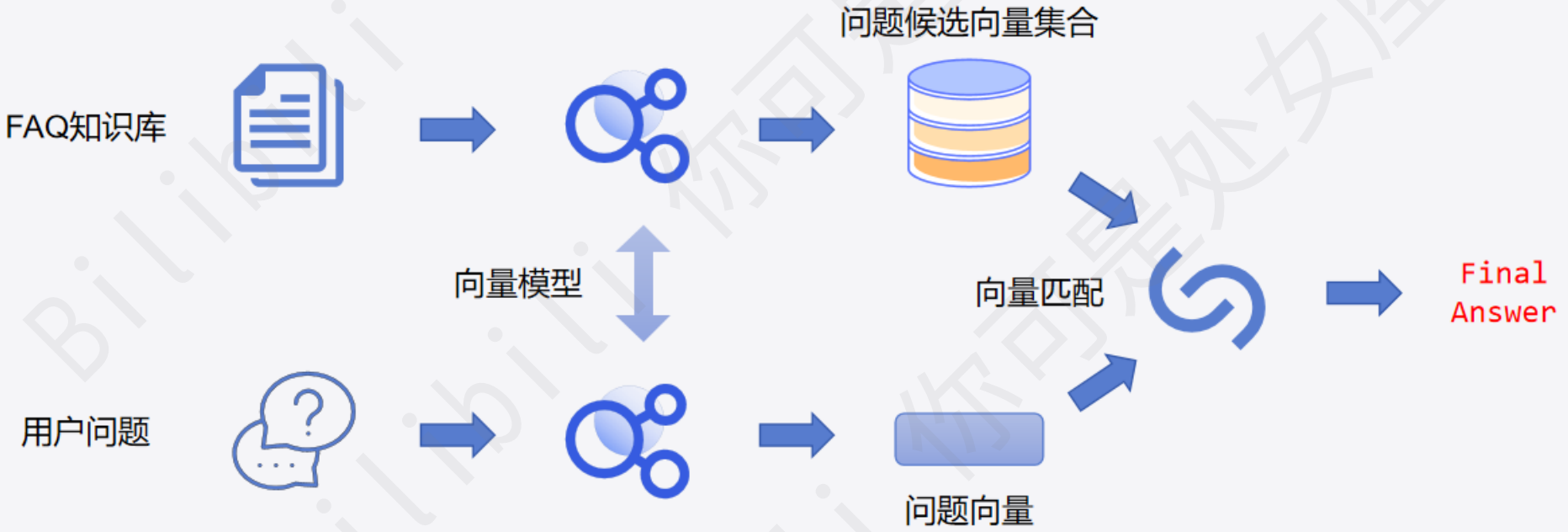
对话机器人介绍

对话机器人简介

- **什么是对话机器人**
  - 对话机器人在本质上是一个用来模拟人类对话或聊天的计算机程序，接收人类的自然语言作为输入，并给出合适的回复
  - 按照任务类型划分，对话机器人简单的可以划分为闲聊机器人、问答机器人、任务型对话机器人
  - 按照答案产生的逻辑划分，对话机器人可以划分为检索式对话机器人和生成式对话机器人
  - 本次课程关注的内容为**基于检索的问答机器人**

基于检索的问答机器人解决方案

- **如何实现基于检索的问答机器人**
  - QQ匹配，取最优结果的Q对应的Answer作为最终结果



代码实战演练（匹配模式/双塔）

- **数据集**
  - https://github.com/SophonPlus/ChineseNlpCorpus
  - 法律知道
- **预训练模型**
  - 上次课程训练的匹配模型
- **额外环境**
  - faiss
  - 安装命令 pip install faiss-cpu

基于检索的问答机器人解决方案

- **进阶版本**
  - 使用向量匹配的模型效果并不好，很难直接取到最优结果，因此引入基于交互策略模型
  - 向量匹配模块又称为召回模块，交互策略的模块又称为排序模块

代码实战演练（匹配模式/双塔）

- **数据集**
  - https://github.com/SophonPlus/ChineseNlpCorpus
  - 法律知道
- **预训练模型**
  - 上次课程训练的匹配模型
  - 上次课程训练的交互模型

07

实战演练之预训练模型

（1）预训练介绍

（2）代码实战，掩码语言模型

（3）代码实战，因果语言模型

预训练介绍

预训练简介

- **什么是预训练**
  - 预训练（Pretrain）是指通过自监督学习从大规模数据中获得与具体任务无关的预训练模型的过程，最终产出为预训练模型（Pretrained Model）。

模型类型	常用预训练模型	适用任务
编码器模型，自编码模型	ALBERT, BERT, DistilBERT, RoBERTa	文本分类、命名实体识别、阅读理解
解码器模型，自回归模型	GPT, GPT-2, Bloom, LLaMA	文本生成
编码器解码器模型，序列到序列模型	BART, T5, Marian, mBART	文本摘要、机器翻译

预训练介绍

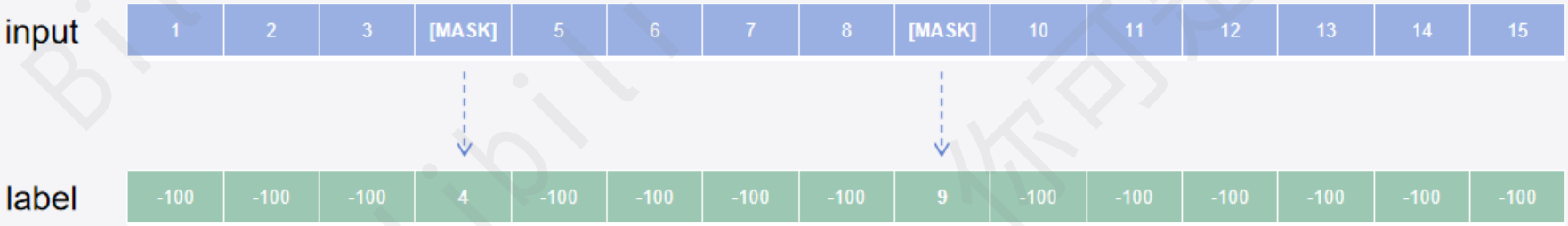
预训练简介

- **预训练任务**
  - 掩码语言模型，自编码模型
    - 将一些位置的token替换成特殊的[MASK]字符，预测这些被替换的字符
  - 因果语言模型，自回归模型
    - 将完整序列输入，基于上文的token预测当前token
  - 序列到序列模型
    - 任务较为多样化，只是采用编码器解码器的方式，预测部分放在解码器中

预训练介绍

预训练简介

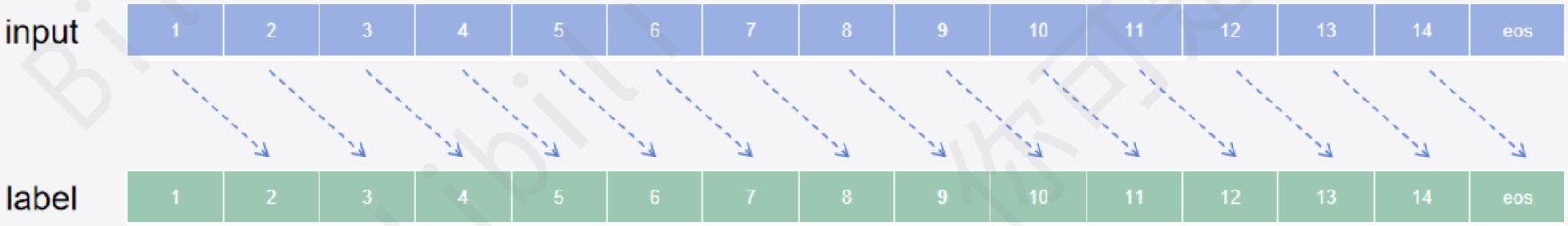
- **预训练任务**
  - 掩码语言模型，自编码模型
    - 将一些位置的token替换成特殊的[MASK]字符，预测这些被替换的字符
    - 只计算掩码部分的loss，其余部分不计算loss



预训练介绍

预训练简介

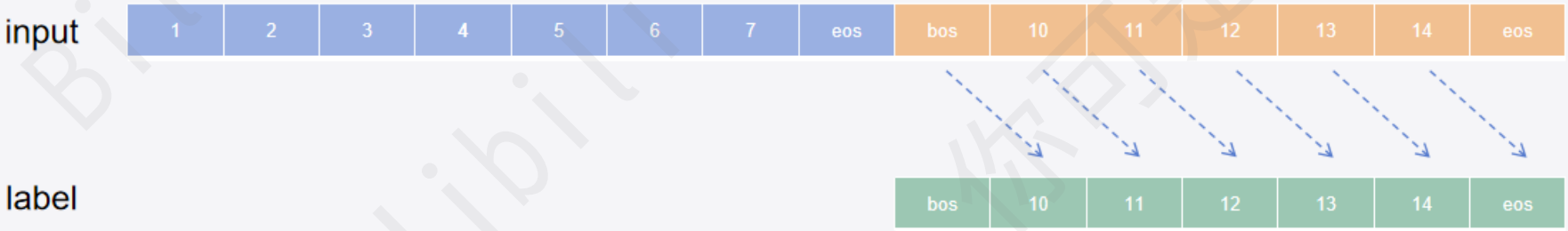
- **预训练任务**
  - 因果语言模型，自回归模型
    - 将完整序列输入，基于上文的token预测当前token
    - 结束位置要有特殊token，eos_token



预训练介绍

预训练简介

- **预训练任务**
  - 序列到序列模型，前缀语言模型（Prefix Language Model）
    - 任务较为多样化，如掩码生成、片段自回归、乱序修正等
    - 采用编码器解码器的方式进行实现，计算解码器部分的loss



代码实战演练（掩码语言模型）

- **数据集**
  - https://huggingface.co/datasets/pleisto/wikipedia-cn-20230720-filtered
  - 百科语料
- **预训练模型**
  - hfl/chinese-macbert-base

代码实战演练（因果语言模型）

- **数据集**
  - https://huggingface.co/datasets/pleisto/wikipedia-cn-20230720-filtered
  - 百科语料
- **预训练模型**
  - Langboat/bloom-389m-zh

08

实战演练之文本摘要

（1）文本摘要介绍

（2）基于Transformers的解决方案

（3）代码实战演练，基于T5模型

（4）代码实战演练，基于GLM模型

文本摘要介绍

文本摘要简介

- **什么是文本摘要任务**
  - 文本摘要（Text Summarization）任务的输入是长的文本文档，任务的目标是将较长的文本转换成简短的摘要，一般来说，生成的简短摘要必须要信息量充足、能够覆盖原文的主要内容。
  - 根据输入文档的数量划分，可以将摘要任务划分为单文档摘要和多文档摘要
  - 根据输入和输出的语言划分，可以将摘要任务划分为单语言摘要、跨语言摘要、多语言摘要
  - 本次课程的关注内容为单文档单语言摘要。

文本摘要介绍

文本摘要简介

- **什么是文本摘要任务**
  - 文档
    - 某卖出自家的马自达后,又使用备用车钥匙将车盗走。因涉嫌盗窃,今天上午,肖某在大兴法院受审。肖某是本市丰台人,大学毕业,无业。检方指控,2013年5月,肖某在大兴区某公司院内,使用其保留的车钥匙将此前卖给被害人的一辆马自达轿车(经鉴定价值11.5元),还有车内价值800元的加油卡、现金600元及台球杆3根及1副太阳镜一并盗走。上午9时,肖某被带进法庭。庭审中,肖某对指控没有异议。肖某的辩护人向法庭提交了一份被告人母亲写的家庭情况说明、被告人母亲的医疗费票据及诊断证明,说明其家庭生活困难,且与被害人达成和解,请求法院轻判。公诉人表示,因被告人与被害人已和解,基于被告人母亲的情况,同意法庭综合考虑量刑。最后陈述阶段,肖某说,“我因为法律意识淡薄,给社会和被害人带来了危害,经过这段时间的改造和反省,我认识到了这一错误。我会尽力补偿被害人的损失,希望法庭给我一次机会,让我用实际行动弥补对家庭的创伤。”该案没有当庭判决。
  - 摘要（标题）
    - 北京大学生卖掉自家私家车,又用备用钥匙偷回来,一并顺走车内现金600元;因家庭困难,法院同意综合量刑。

文本摘要简介

- **评价指标**
  - Rouge
    - Rouge-1、Rouge-2、Rouge-L
    - 分别基于1-gram、2-gram、LCS
  - 示例

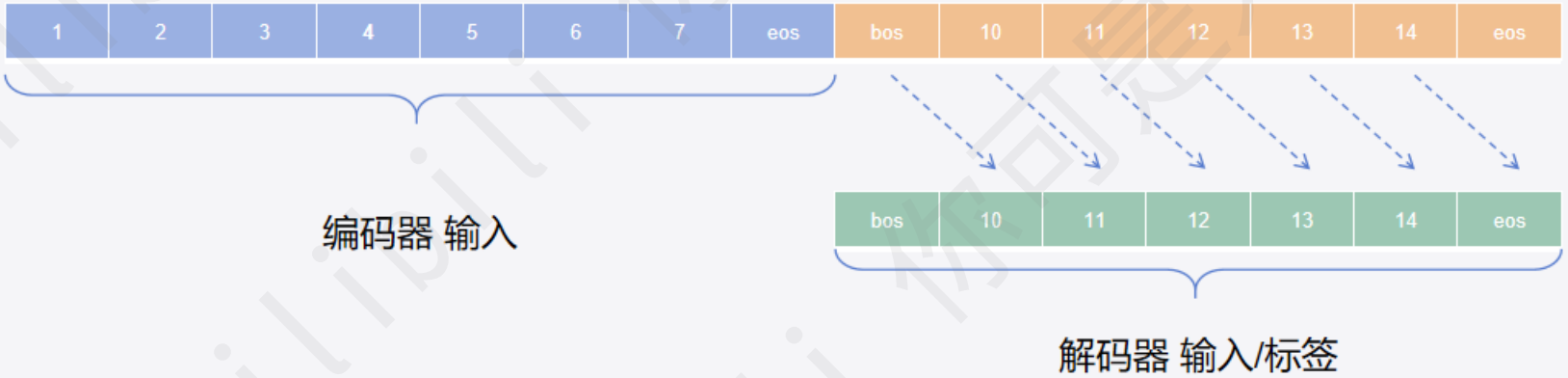
原始文本	1-gram	2-gram
今天不错	今 天 不 错	今天 天不 不错
今天太阳不错	今 天 太 阳 不 错	今天 天太 太阳 阳不 不错

    - Rouge-1 P = 4/4， R = 4/6，F = 2*P*R / (P + R)
    - Rouge-2 P = 2/3， R = 2/5，F = 2*P*R / (P + R)
    - Rouge-L P = 4/4， R = 4/6，F = 2*P*R / (P + R)

基于Transformers的解决方案介绍

序列到序列模型

- **数据处理与模型原理**
  - 数据处理
    - input和labels分开处理，labels的最后一定是eos_token
    - labels不仅是标签，还是解码器的输入



基于Transformers的解决方案介绍

序列到序列模型

- **数据处理与模型原理**
  - XXForConditionalGeneration

```python
def __init__(self, config: T5Config):
    super().__init__(config)
    self.model_dim = config.d_model

    self.shared = nn.Embedding(config.vocab_size, config.d_model)

    encoder_config = copy.deepcopy(config)
    encoder_config.is_decoder = False
    encoder_config.use_cache = False
    encoder_config.is_encoder_decoder = False
    self.encoder = T5Stack(encoder_config, self.shared)

    decoder_config = copy.deepcopy(config)
    decoder_config.is_decoder = True
    decoder_config.is_encoder_decoder = False
    decoder_config.num_layers = config.num_decoder_layers
    self.decoder = T5Stack(decoder_config, self.shared)

    self.lm_head = nn.Linear(config.d_model, config.vocab_size, bias=False)

    # Initialize weights and apply final processing
    self.post_init()

    # Model parallel
    self.model_parallel = False
    self.device_map = None
```

基于Transformers的解决方案介绍

序列到序列模型

- **数据处理与模型原理**
  - XXForConditionalGeneration

```
# Encode if needed (training, first prediction pass)
if encoder_outputs is None:
    # Convert encoder inputs in embeddings if needed
    encoder_outputs = self.encoder(
        input_ids=input_ids,
        attention_mask=attention_mask,
        inputs_embeds=inputs_embeds,
        head_mask=head_mask,
        output_attentions=output_attentions,
        output_hidden_states=output_hidden_states,
        return_dict=return_dict,
    )
elif return_dict and not isinstance(encoder_outputs, BaseModelOutput):
    encoder_outputs = BaseModelOutput(
        last_hidden_state=encoder_outputs[0],
        hidden_states=encoder_outputs[1] if len(encoder_outputs) > 1 else None,
        attentions=encoder_outputs[2] if len(encoder_outputs) > 2 else None,
    )

hidden_states = encoder_outputs[0]
```

基于Transformers的解决方案介绍

序列到序列模型

- **数据处理与模型原理**
  - XXForConditionalGeneration

```
# Decode
decoder_outputs = self.decoder(
    input_ids=decoder_input_ids,
    attention_mask=decoder_attention_mask,
    inputs_embeds=decoder_inputs_embeds,
    past_key_values=past_key_values,
    encoder_hidden_states=hidden_states,
    encoder_attention_mask=attention_mask,
    head_mask=decoder_head_mask,
    cross_attn_head_mask=cross_attn_head_mask,
    use_cache=use_cache,
    output_attentions=output_attentions,
    output_hidden_states=output_hidden_states,
    return_dict=return_dict,
)

sequence_output = decoder_outputs[0]
```

基于Transformers的解决方案介绍

序列到序列模型

- **数据处理与模型原理**
  - XXForConditionalGeneration

```
lm_logits = self.lm_head(sequence_output)

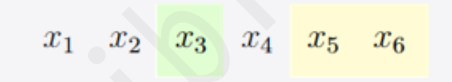
loss = None
if labels is not None:
    loss_fct = CrossEntropyLoss(ignore_index=-100)
    # move labels to correct device to enable PP
    labels = labels.to(lm_logits.device)
    loss = loss_fct(lm_logits.view(-1, lm_logits.size(-1)), labels.view(-1))
```

代码实战演练（基于T5模型）

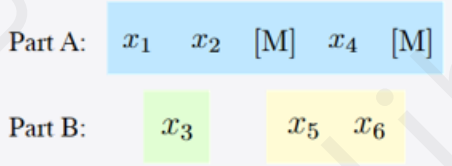
- **数据集**
  - https://huggingface.co/datasets/supremezxc/nlpcc_2017
  - 新闻标题生成
- **预训练模型**
  - Langboat/mengzi-t5-base
- **依赖库**
  - pip install rouge-chinese

前缀语言模型

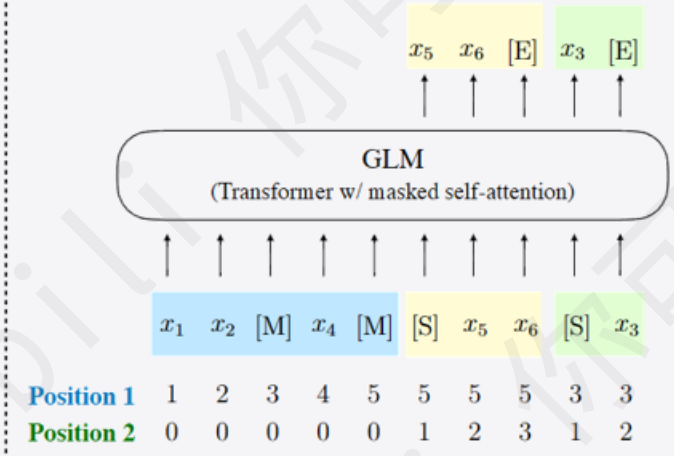
- **数据处理与模型原理**
  - 模型原理
    - 只使用编码器，借助注意力掩码实现编码器解码器的效果，只计算目标部分损失



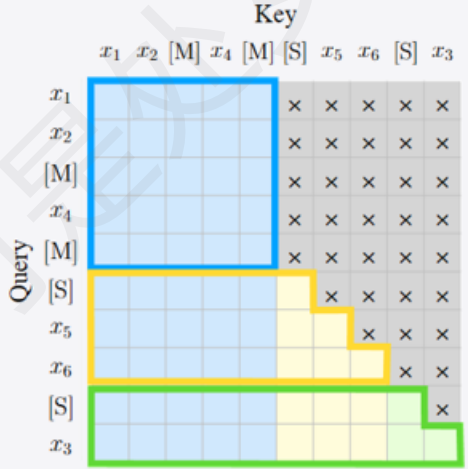
(a) Sample spans from the input text



(b) Divide the input into Part A / Part B



(c) Generate the Part B spans autoregressively

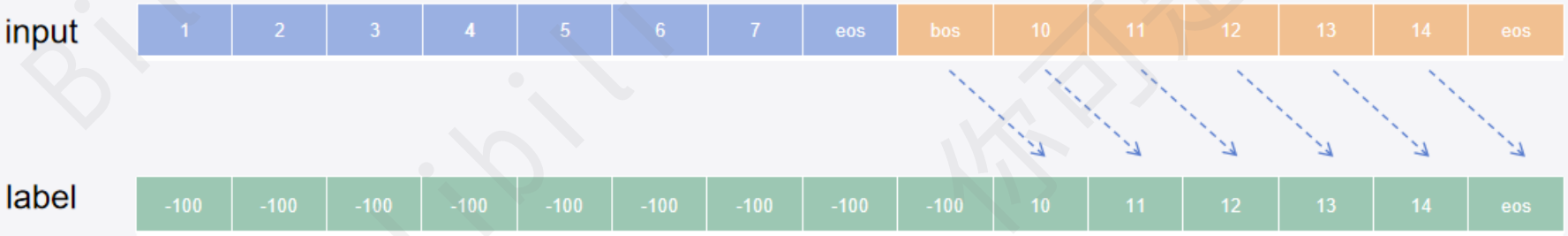


(d) Self-attention mask

基于Transformers的解决方案介绍

前缀语言模型

- **数据处理与模型原理**
  - 模型原理
    - 只使用解码器，借助注意力掩码实现编码器解码器的效果，只计算目标部分损失
  - 数据处理
    - input和labels合并在一起处理，labels的最后一定是eos_token



代码实战演练（基于GLM模型）

- **数据集**
  - https://huggingface.co/datasets/supremezxc/nlpcc_2017
  - 新闻标题生成
- **预训练模型**
  - THUDM/glm-large-chinese
- **依赖库**
  - pip install rouge-chinese

09

实战演练之
生成式对话机器人

（1）对话机器人简介

（2）代码实战演练，基于BLOOM

（3）常见解码参数介绍

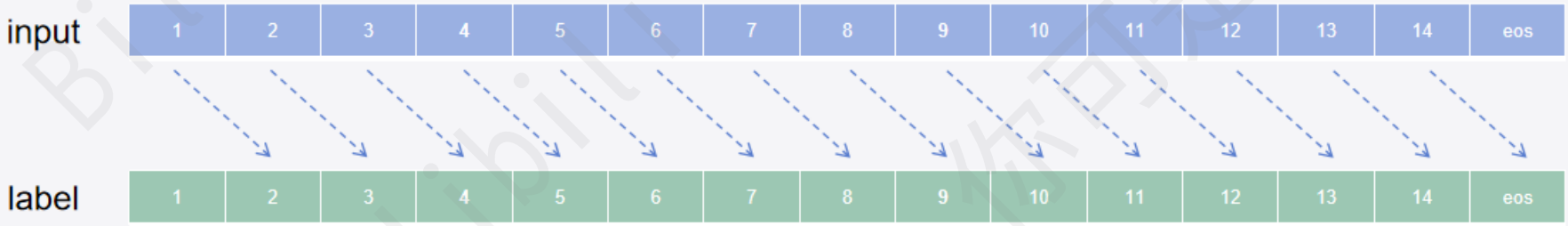
对话机器人介绍

对话机器人简介

- **什么是对话机器人**
  - 对话机器人在本质上是一个用来模拟人类对话或聊天的计算机程序，接收人类的自然语言作为输入，并给出合适的回复
  - 按照任务类型划分，对话机器人简单的可以划分为闲聊机器人、问答机器人、任务型对话机器人
  - 按照答案产生的逻辑划分，对话机器人可以划分为检索式对话机器人和生成式对话机器人
  - 本次课程关注的内容为**基于生成式的问答机器人**

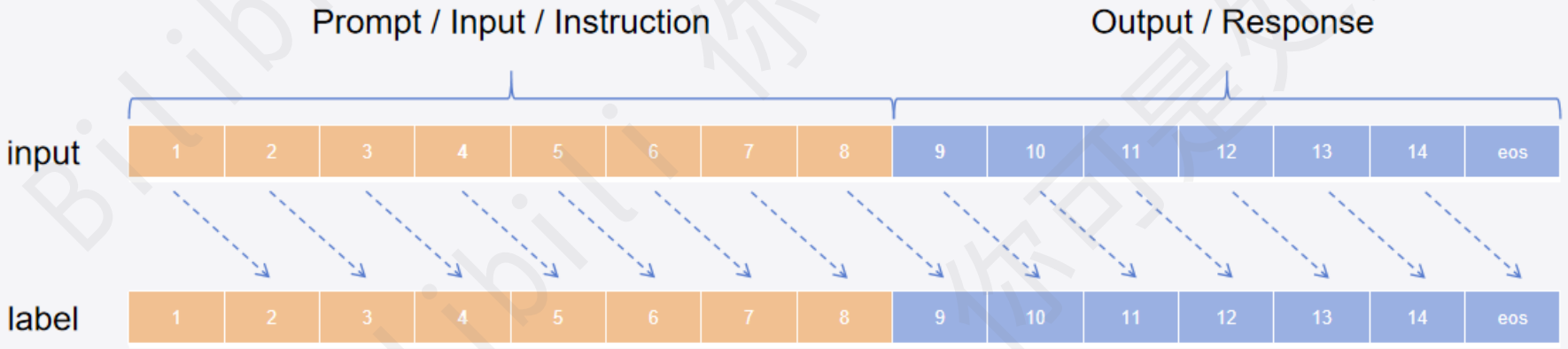
预训练简介

- **预训练任务**
  - 因果语言模型，自回归模型
    - 将完整序列输入，基于上文的token预测当前token
    - 结束位置要有特殊token，eos_token



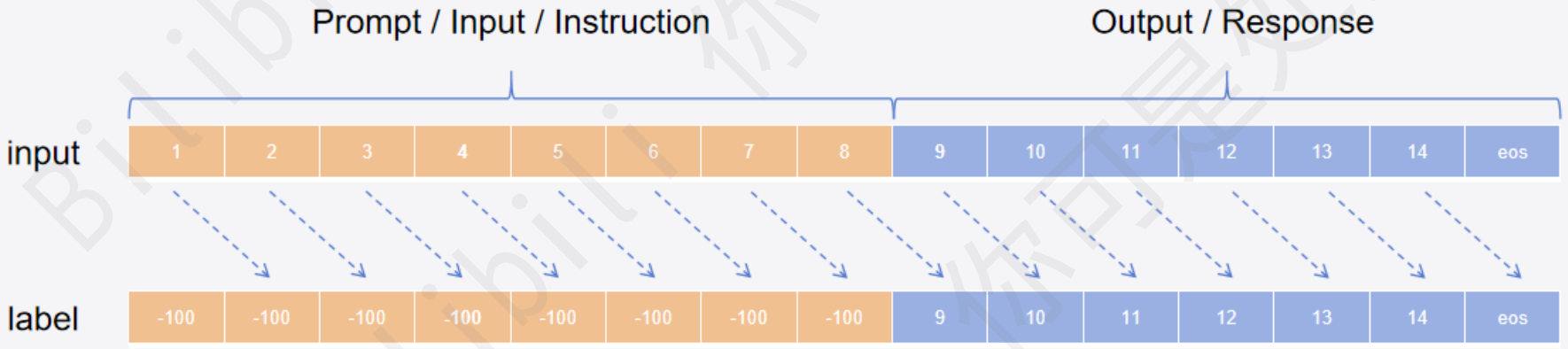
对话机器人解决方案

- **指令微调**
  - 指令微调的方式，赋予回答问题的能力
    - 多类型的任务共同学习，能够解决不同的任务



对话机器人解决方案

- **指令微调**
  - 指令微调的方式，赋予回答问题的能力
    - 训练单轮问答模型，计算Loss时只计算Output部分



对话机器人解决方案

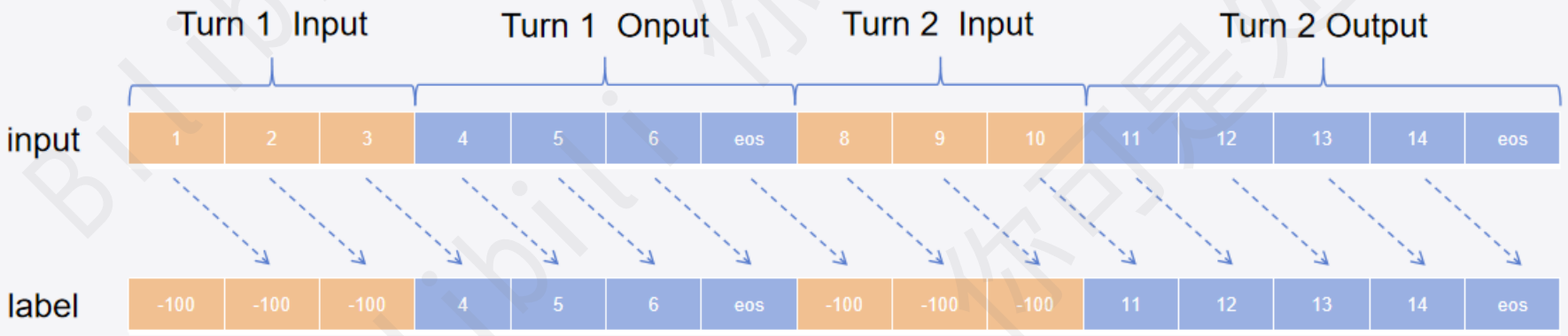
- **指令微调**
  - 指令微调的方式，赋予回答问题的能力
    - 多轮如何计算？

对话机器人解决方案

- **指令微调**
  - 指令微调的方式，赋予回答问题的能力
    - 方式一 计算最后一轮Output的Loss，效率较低

对话机器人解决方案

- **指令微调**
  - 指令微调的方式，赋予回答问题的能力
    - 方式二 计算每一轮Output的Loss，效率更高



代码实战演练（基于Bloom模型）

- **数据集**
  - https://huggingface.co/datasets/shibing624/alpaca-zh
  - 指令微调
- **预训练模型**
  - Langboat/bloom-389m-zh

常见解码参数介绍

- **常用推理参数**
  - 长度控制
    - min/max_new_tokens 最小/最大生成的长度
    - min/max_length 序列整体的最小/最大长度
  - 解码策略
    - do_sample 是否启用采样的生成方式
    - num_beams beam_search的大小
  - 采样参数
    - temperature 默认1.0，即原始分布，低于1.0会使得分布更尖锐，高于1.0会使得分布更均匀
    - top_k 将词概率从大到小排列，将采样限制在前K个词
    - top_p 将词概率从大到小排列，将采样限制在前N个词，条件是这N个词的概率超过top_p的值
  - 惩罚项
    - repetition_penalty 重复惩罚项，实现原理是降低已经出现过的token的概率

实战演练篇

完结

小结与后期规划

- **实战演练篇小结**
  - 基于Transformers的NLP任务解决方案
    - Transformers解决方案：Tokenizer、Model、Datasets、Evaluate、Trainer
    - 低显存训练大模型：梯度累加、梯度检查点、优化器、参数冻结
    - 自然语言理解：文本分类、命名实体识别、机器阅读理解、多项选择、文本相似度、检索机器人
    - 自然语言生成：掩码语言模型、因果语言模型、文本摘要、生成式对话机器人
    - 所使用模型：MacBERT-base（Masked LM）、T5（Seq2Seq LM）、GLM（Prefix LM）、Bloom-389m-zh（Causal LM）
- **后期规划**
  - 模型高效训练
    - 参数高效微调：基于PEFT库的参数高效微调，介绍PEFT库的使用
    - 分布式训练：基于Accelerate的单机多卡训练，介绍Accelerate库的使用

高效微调篇

基于PEFT的低显存模型微调解决方案

你可是处女座啊

目录

01

参数高效微调简介

（1）什么是参数高效微调

（2）常见的参数高效微调方法

（3）BitFit 代码实战

参数高效微调介绍

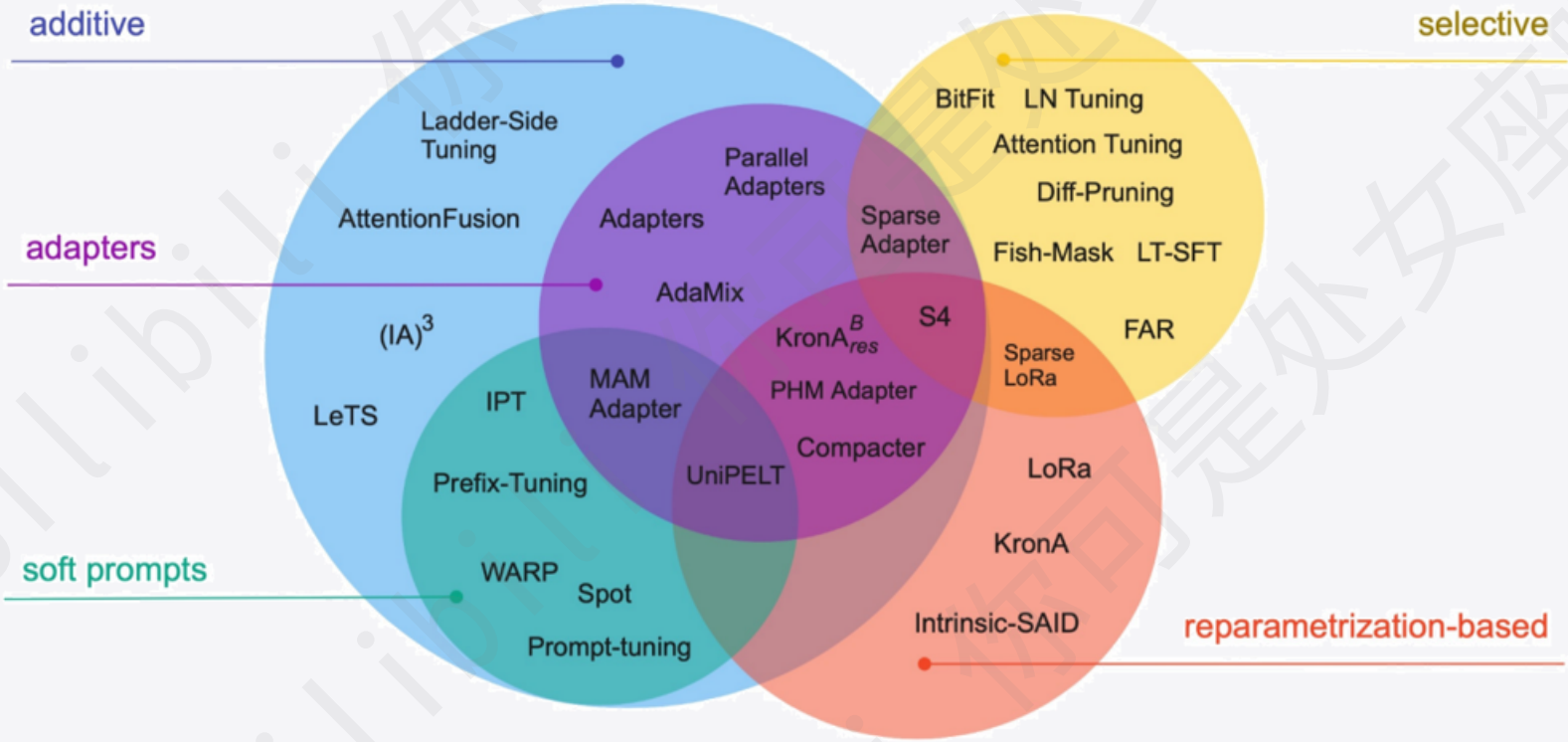
- **背景**
  - 2018年BERT问世后，NLP任务的主流范式：预训练语言模型 + 微调
    - 自编码模型、自回归模型、编码器解码器模型
    - 对于大多数模型的自我对比，尺寸越大，效果越好
    - 预训练语言模型的趋势：大 → 更大 → 再大
    - BERT-Base只有0.1B，而现在意义上的大模型，起步就要6、7B
  - 更大的模型带来了更好的性能，但是也带来了一些问题
    - 以传统微调的方式对模型进行全量参数更新会消耗巨大的计算与存储资源
    - 个人层面，很难拥有充足的资源进行训练

参数高效微调介绍

- **参数高效微调**
  - 在上述的情况下，参数高效微调（Parameter-efficient fine-tuning，PEFT）应运而生
  - 参数高效微调方法仅对模型的一小部分参数（这一小部分可能是模型自身的，也可能是外部引入的）进行训练，便可以为模型带来显著的性能变化，一些场景下甚至不输于全量微调，颇有一种四两拨千斤的感觉
  - 由于训练一小部分参数，极大程度降低了训练大模型的算力需求，不需要多机多卡，单卡即可完成对一些大模型的训练，不仅如此，少量的训练参数对存储的要求同样降低了很多，大多数的参数高效微调方法只需要保存训练部分的参数，与动辄几十GB的原始大模型相比，几乎可以忽略

为什么需要参数高效微调

常见的参数高效微调方法



Lialin V, Deshpande V, Rumshisky A. Scaling down to scale up: A guide to parameter-efficient fine-tuning[J]. arXiv preprint arXiv:2303.15647, 2023.

代码实战演练（基于Bloom模型）

- **数据集**
  - https://huggingface.co/datasets/shibing624/alpaca-zh
  - 指令微调
- **预训练模型**
  - Langboat/bloom-1b4-zh
- **参数高效微调方法**
  - BitFit

02

Prompt-Tuning

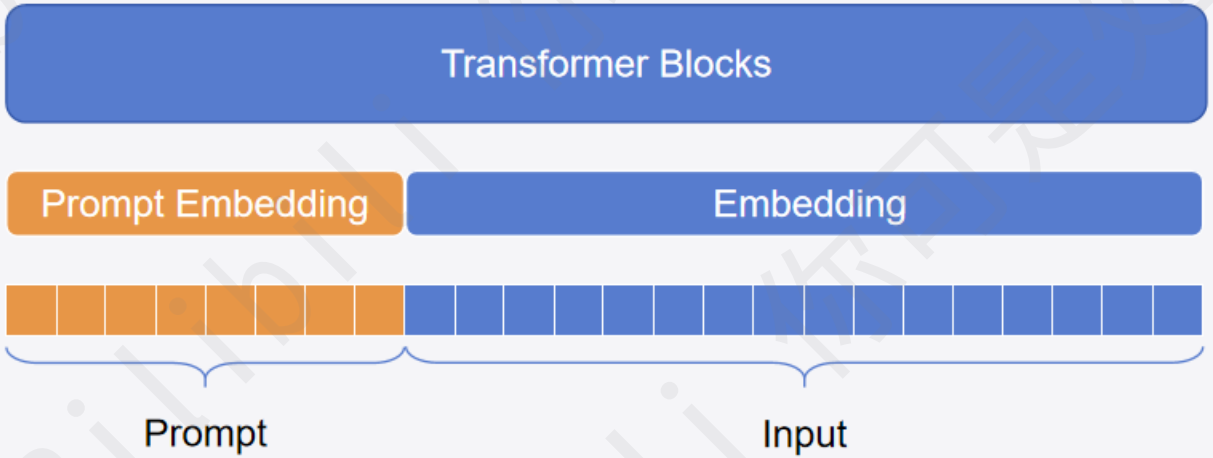
（1）Prompt-Tuning 原理介绍

（2）PEFT库安装与简介

（3）Prompt-Tuning 代码实战

Prompt-Tuning 原理介绍

- **Prompt-Tuning**
  - Prompt-Tuning的思想：冻结主模型全部参数，在训练数据前加入一小段Prompt，只训练Prompt的表示层，即一个Embedding模块。其中，Prompt又存在两种形式，一种是hard prompt，一种是soft prompt。



PEFT环境配置

环境配置

- **PEFT安装**
  - https://github.com/huggingface/peft
  - pip install peft
- **环境更新**
  - transformers 4.33.1
  - peft 0.5.0
  - accelerate 0.22.0

Models support matrix

Causal Language Modeling

Model	LoRA	Prefix Tuning	P-Tuning	Prompt Tuning	IA3
GPT-2	✅	✅	✅	✅	✅
Bloom	✅	✅	✅	✅	✅
OPT	✅	✅	✅	✅	✅
GPT-Neo	✅	✅	✅	✅	✅
GPT-J	✅	✅	✅	✅	✅
GPT-NeoX-20B	✅	✅	✅	✅	✅
LLaMA	✅	✅	✅	✅	✅
ChatGLM	✅	✅	✅	✅	✅

代码实战演练（基于Bloom模型）

- **数据集**
  - https://huggingface.co/datasets/shibing624/alpaca-zh
  - 指令微调
- **预训练模型**
  - Langboat/bloom-1b4-zh
- **参数高效微调方法**
  - Prompt tuning

03

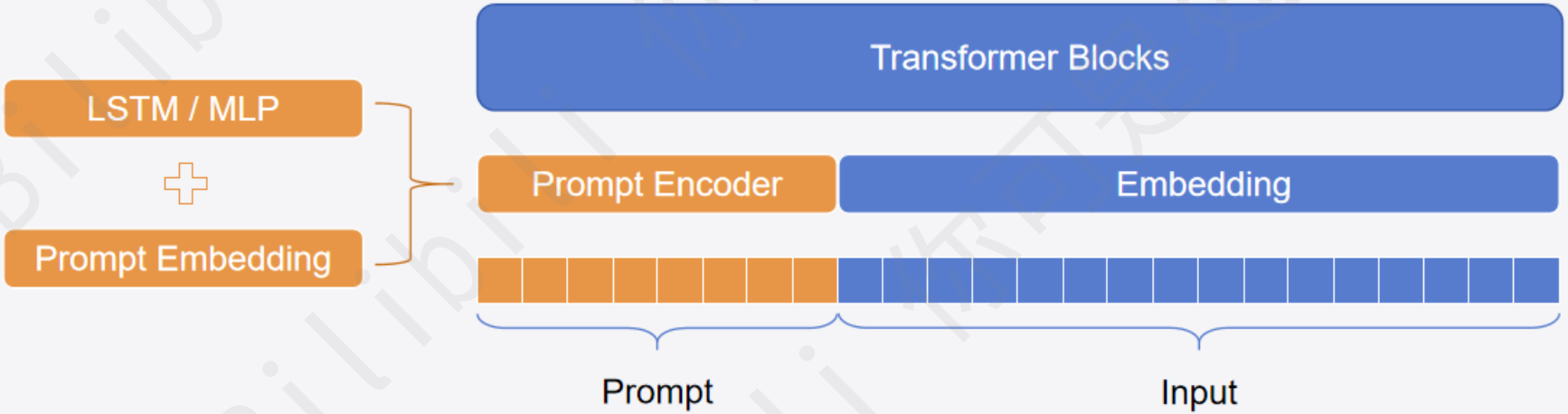
P-Tuning

（1）P-Tuning 原理介绍

（2）P-Tuning 代码实战

P–Tuning 原理介绍

- **P-Tuning**
  - P-Tuning的思想：在Prompt-Tuning的基础上，对Prompt部分进行进一步的编码计算，加速收敛。具体来说，PEFT中支持两种编码方式，一种是LSTM，一种是MLP。与Prompt-Tuning不同的是，Prompt的形式只有Soft Prompt。



PEFT环境配置

环境配置

- **PEFT安装**
  - https://github.com/huggingface/peft
  - pip install peft
- **环境更新**
  - transformers 4.33.1
  - peft 0.5.0
  - accelerate 0.22.0

代码实战演练（基于Bloom模型）

- **数据集**
  - https://huggingface.co/datasets/shibing624/alpaca-zh
  - 指令微调
- **预训练模型**
  - Langboat/bloom-1b4-zh
- **参数高效微调方法**
  - P-tuning

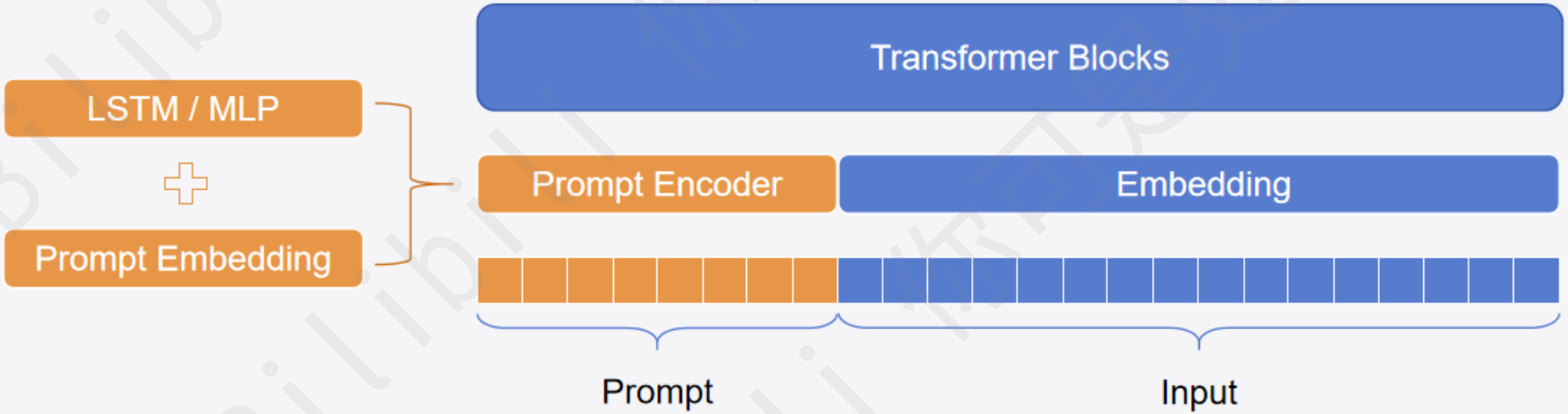
04 Prefix-Tuning

（1）Prefix-Tuning 原理介绍

（2）Prefix-Tuning 代码实战

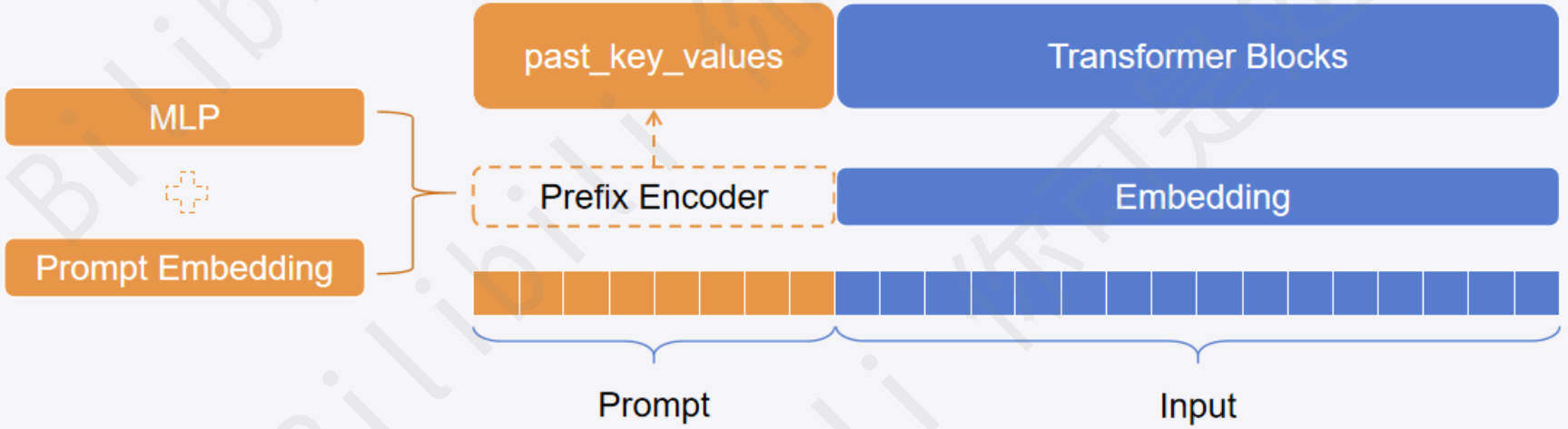
P–Tuning 回顾

- **P-Tuning**
  - P-Tuning的思想：在Prompt-Tuning的基础上，对Prompt部分进行进一步的编码计算，加速收敛。具体来说，PEFT中支持两种编码方式，一种是LSTM，一种是MLP。与Prompt-Tuning不同的是，Prompt的形式只有Soft Prompt。



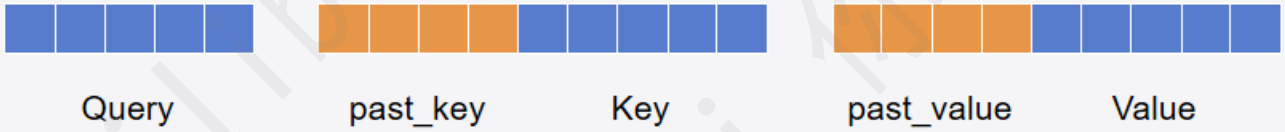
Prefix–Tuning 原理介绍

- **Prefix-Tuning**
  - Prefix-Tuning的思想：相较于Prompt-Tuning和P-tuning，Prefix-Tuning不再将Prompt加在输入的Embedding层，而是将其作为可学习的前缀，放置在Transformer模型中的每一层中，具体表现形式为past_key_values。



Prefix-Tuning 原理介绍

- **如何理解past_key_values**
  - past_key_values：Transformer模型中历史计算过的key和value的结果，最早是用于生成类模型解码加速，解码逻辑是根据历史输入，每次预测一个新的token，然后将新的token加入输入，再预测下一个token。这个过程中，会存在大量的重复计算，因此可以将key和value的计算结果缓存，作为past_key_values输入到下一次的计算中，这一技术又被称之为kv_cache。
  - Prefix-Tuning中，就是通过past_key_values的形式将可学习的部分放到了模型中的每一层，这部分内容又被称之为前缀



PEFT环境配置

环境配置

- **PEFT安装**
  - https://github.com/huggingface/peft
  - pip install peft
- **环境更新**
  - transformers 4.33.2
  - peft 0.5.0
  - accelerate 0.22.0

代码实战演练（基于Bloom模型）

- **数据集**
  - https://huggingface.co/datasets/shibing624/alpaca-zh
  - 指令微调
- **预训练模型**
  - Langboat/bloom-1b4-zh
- **参数高效微调方法**
  - Prefix-Tuning

05

Lora

（1）Lora 原理介绍

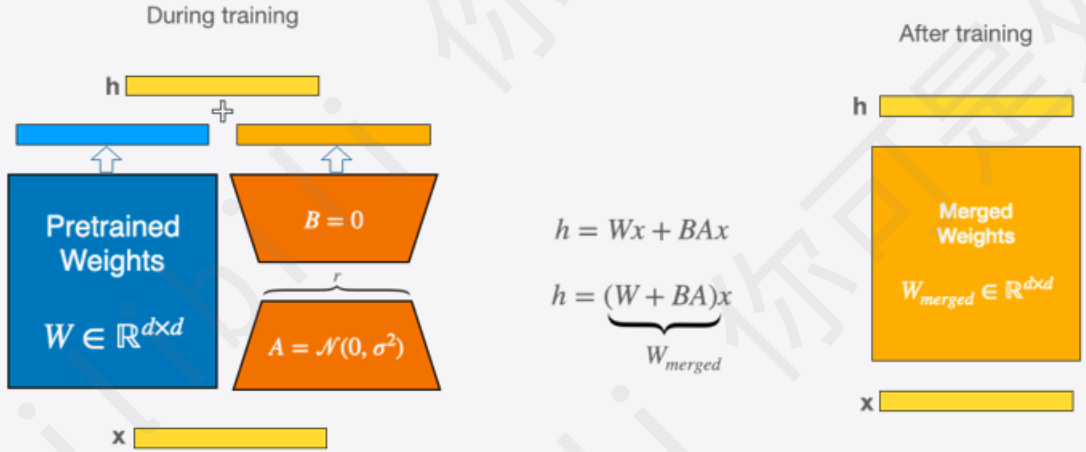
（2）Lora 代码实战

Lora 原理介绍

- **Lora**
  - Lora的思想：
    - 预训练模型中存在一个极小的内在维度，这个内在维度是发挥核心作用的地方。
    - 在继续训练的过程中，权重的更新依然也有如此特点，即也存在一个内在维度（内在秩）。
    - 权重更新：$W = W + \triangle W$
    - 因此，可以通过矩阵分解的方式，将原本要更新的大的矩阵变为两个小的矩阵。
    - 权重更新：$W = W + \triangle W = W + BA$
    - 具体做法，即在矩阵计算中增加一个旁系分支，旁系分支由两个低秩矩阵A和B组成。

Lora 原理介绍

- **Lora**
  - 训练时，输入分别与原始权重和两个低秩矩阵进行计算，共同得到最终结果，优化则仅优化A和B。
  - 训练完成后，可以将两个低秩矩阵与原始模型中的权重进行合并，合并后的模型与原始模型无异，避免了推理期间Prompt系列方法带来的额外计算量



PEFT环境配置

环境配置

- **PEFT安装**
  - https://github.com/huggingface/peft
  - pip install peft
- **环境更新**
  - transformers 4.34.0
  - peft 0.5.0
  - accelerate 0.23.0

代码实战演练（基于Bloom模型）

- **数据集**
  - https://huggingface.co/datasets/shibing624/alpaca-zh
  - 指令微调
- **预训练模型**
  - Langboat/bloom-1b4-zh
- **参数高效微调方法**
  - Lora

06

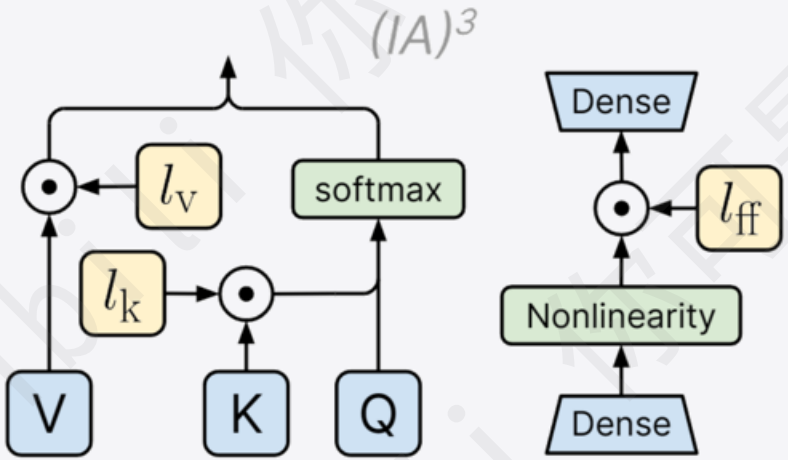
IA3

（1）IA3 原理介绍

（2）IA3 代码实战

IA3 原理介绍

- **IA3，Infused Adapter by Inhibiting and Amplifying Inner Activations**
  - IA3的思想：抑制和放大内部激活，通过可学习的向量对激活值进行抑制或放大。具体来说，会对K、V、FFN三部分的值进行调整，训练过程中同样冻结原始模型的权重，只更新可学习的部分向量部分。训练完成后，与Lora类似，也可以将学习部分的参数与原始权重合并，没有额外推理开销。



代码实战演练（基于Bloom模型）

- **数据集**
  - https://huggingface.co/datasets/shibing624/alpaca-zh
  - 指令微调
- **预训练模型**
  - Langboat/bloom-1b4-zh
- **参数高效微调方法**
  - IA3

07

PEFT进阶操作

（1）自定义模型适配

（2）多适配器加载与切换

（3）禁用适配器

PEFT进阶操作

- PEFT进阶操作
  - 如何为自定义的模型适配参数高效微调
    - 自定义模型适配
  - 一个主模型，多个适配器的情况如何使用
    - 多适配器加载与切换
  - 如何获取原始模型的输出结果
    - 禁用适配器

课程回顾

- **参数高效微调**
  - 参数高效微调方法：
    - BitFit、Prompt-Tuning、P-Tuning、Prefix-Tuning、LoRA、IA3
  - 如何使用PEFT库进行参数高效微调
    - 训练
      - Step1 XXConfig(task_type=XXX)
      - Step2 get_peft_model(model, config)
    - 加载
      - PeftModel.from_pretrained(model, model_id)
      - peft_model.merge_and_unload()

高效微调篇

完结

下一步规划

- **如何训练更大的模型**
  - 24G的显存，能不能训练7B的模型？
    - Llama2-7B
    - FP32：7 * 4 = 28G
  - 如何降低显存占用？

下一步规划

- **如何训练更大的模型**
  - 24G的显存，能不能训练7B的模型？
    - Llama2-7B
    - FP32：7 * 4 = 28G
  - 如何降低显存占用？

低精度训练

低精度训练篇

基于bitsandbytes的低精度训练方法

你可是处女座啊

目录

01 低精度训练与模型下载

（1）低精度训练背景介绍

（2）基于modelscope的大模型下载

背景介绍

- **大模型训练的难点是什么**
  - 计算效率
    - 大模型的训练依赖于海量的训练数据，海量的训练数据带来了海量的计算需求
  - 显存效率
    - 大模型主要体现在模型参数规模上，参数规模逐渐变大的模型对显存的依赖逐渐加剧

背景介绍

- **大模型训练的难点是什么**
  - 计算效率
    - 大模型的训练依赖于海量的训练数据，海量的训练数据带来了海量的计算需求
  - 显存效率
    - 大模型主要体现在模型参数规模上，参数规模逐渐变大的模型对显存的依赖逐渐加剧

显存问题是我们要解决的主要问题！

背景介绍

- **模型训练的显存占用**
  - 模型权重
    - 4Bytes * 模型参数量
  - 优化器状态
    - 8Bytes * 模型参数量，对于常用的AdamW优化器而言
  - 梯度
    - 4Bytes * 模型参数量
  - 前向激活值
    - 取决于序列长度、隐层维度、Batch大小等多个因素

背景介绍

- **模型训练的显存占用**
  - 模型权重
    - 4Bytes * 模型参数量
  - 优化器状态
    - 8Bytes * 模型参数量，对于常用的AdamW优化器而言
  - 梯度
    - 4Bytes * 模型参数量
  - 前向激活值
    - 取决于序列长度、隐层维度、Batch大小等多个因素

如何降低训练时的显存占用？

背景介绍

- **如何降低训练时显存占用**
  - 实战演练篇——4G显存，0.3B模型
    - 梯度累计
    - 梯度检查点
    - 优化器配置
    - 输入数据长度
    - 冻结模型参数
  - 参数高效微调篇——8G显存，1.4B模型
    - 参数高效微调（Prompt-Tuning、LoRA等）

背景介绍

- **如何降低训练时显存占用**
  - 实战演练篇——4G显存，0.3B模型
    - 梯度累计
    - 梯度检查点
    - 优化器配置
    - 输入数据长度
    - 冻结模型参数
  - 参数高效微调篇——8G显存，1.4B模型
    - 参数高效微调（Prompt-Tuning、LoRA等）

训练过程中，模型本身的显存占用并未发生变化！

背景介绍

- **如何降低训练时显存占用**
  - 实战演练篇——4G显存，0.3B模型
    - 梯度累计
    - 梯度检查点
    - 优化器配置
    - 输入数据长度
    - 冻结模型参数
  - 参数高效微调篇——8G显存，1.4B模型
    - 参数高效微调（Prompt-Tuning、LoRA等）

如何降低训练过程中模型本身的显存占用？

背景介绍

- **模型自身的显存占用**
  - 显存占用
    - 显存占用 ~= 4Bytes * 模型参数量
  - 如何降低显存占用
    - 参数量不变的情况下，降低模型中每个参数占用的字节数即可降低模型的显存占用
  - 如何降低参数占用的字节数
    - 默认的数值精度为单精度fp32，32bits，4Bytes
    - 使用低精度的数据类型表示即可
    - 常见的低精度数据类型：fp16（half、半精度）、bfloat16、int8、fp4、nf4

模型下载

- **基于modelscope的大模型下载方式**
  - modelscope 环境配置
    - conda create -n ms python=3.9
    - conda activate ms
    - pip install modelscope jupyterlab
  - 模型下载代码
    - from modelscope.hub.snapshot_download import snapshot_download
    - snapshot_download(model_id="XXX", cache_dir="XXX")

02

半精度模型训练实战

（1）半精度简介

（2）代码实战演练（上，LlaMA2）

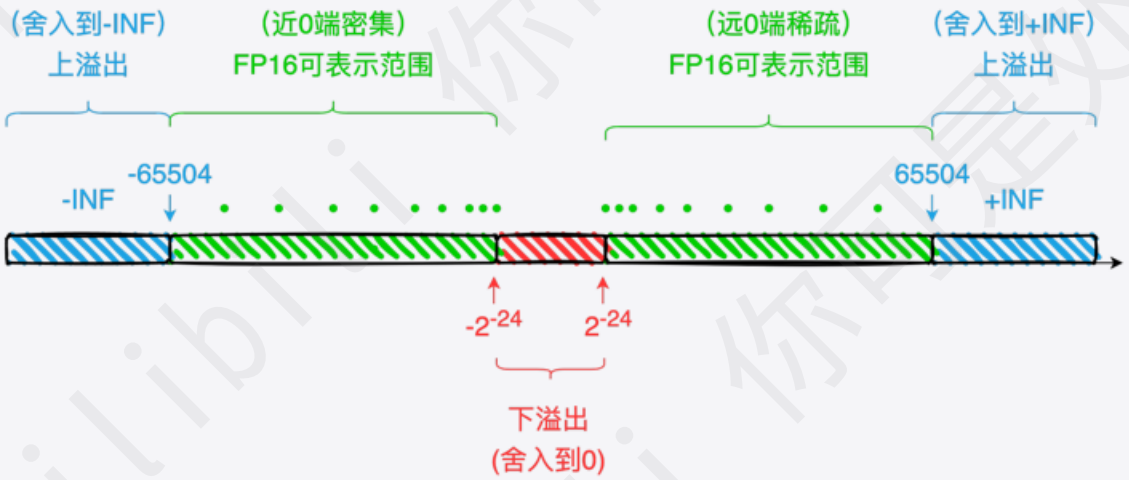
（3）代码实战演练（下，ChatGLM2）

半精度介绍

- **什么是半精度**
  - 半精度FP16（half precision）是一种浮点数格式，它使用16bit表示一个数字（2个字节）
  - 在训练过程中，启用半精度训练可以有效节约显存，并提升计算速度

半精度介绍

- **什么是半精度**
  - 半精度FP16（half precision）是一种浮点数格式，它只占用16位（2个字节）
  - 在计算过程中的问题，可能存在溢出问题和舍入问题，可以使用bf16替代



半精度模型训练

如何启用半精度训练

- **如何启用半精度训练**
  - 方式一
    - 模型加载后调用half方法将单精度模型转为半精度模型
    - model = model.half()
  - 方式二（推荐）
    - 模型加载时，指定torch_dtype参数为torch.half/torch.float16
    - XXModel.from_pretrained(..., torch_dtype=torch.half)

代码实战演练（上，基于LlaMA2）

- **LlaMA2 简介**
  - Meta开源的预训练大模型
  - 基于自回归架构的预训练模型
  - 公开可以拿到的模型尺寸包括：7B、13B、70B
  - 丰富的中文拓展版本，社区活跃

代码实战演练（上，基于LlaMA2）

- **数据集**
  - https://huggingface.co/datasets/shibing624/alpaca-zh
  - 指令微调
- **预训练模型**
  - LlaMA2-7b
- **参数高效微调方法**
  - Lora

代码实战演练

模型训练细节

- **LlaMA2 模型训练细节**
  - LlaMA2模型分词器的padding_side要设置为right，否则可能不收敛
  - LlaMA2模型的分词器词表的问题，要适当调整数据的最大长度，保证数据内容的完整性
  - LlaMA2模型加载时，需要指定torch_dtype为半精度，否则模型将按照fp32进行加载
  - 当启用gradient_checkpoint训练时，需要一并调用model.enable_input_require_grads()方法
  - 当完全采用fp16半精度进行训练且采用adam优化器时，需要调整优化器中的adam_epsilon的值，否则模型无法收敛

模型训练细节

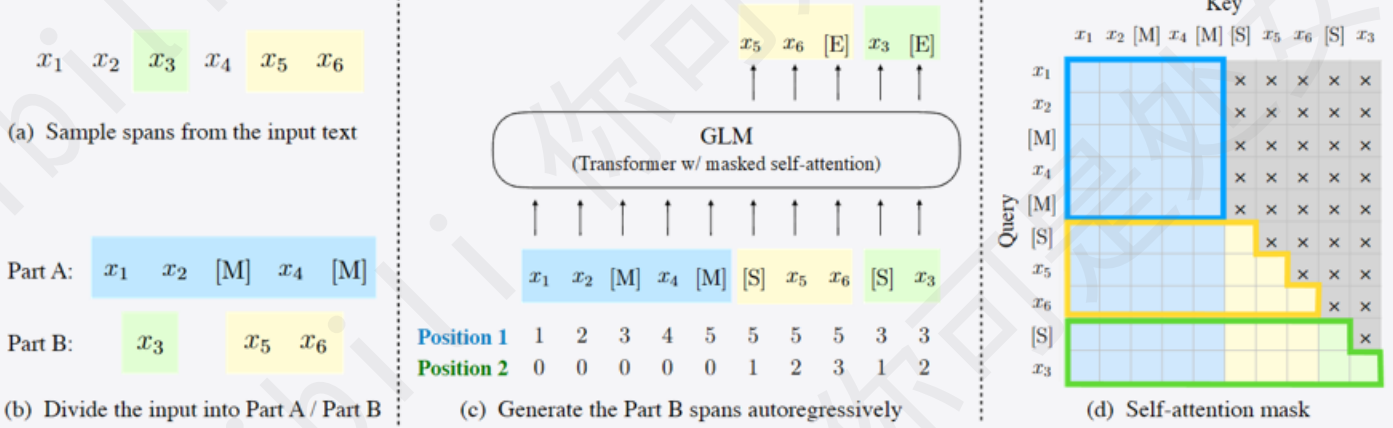
- **LlaMA2 模型训练细节补充**
  - LlaMA2模型分词器会将非单独存在的eos_token切开，因此对于eos_token要单独处理，否则训练后的模型在预测时不知道何时停止
  - 半精度训练时，正确加入eos_token后，要将pad_token_id也置为eos_token_id，否则模型通用无法收敛

代码实战演练（下，基于ChatGLM3）

- **数据集**
  - https://huggingface.co/datasets/shibing624/alpaca-zh
  - 指令微调
- **预训练模型**
  - ChatGLM3-6B-base
- **参数高效微调方法**
  - Lora

ChatGLM

- **GLM和ChatGLM**
  - GLM模型
    - 训练方式 Prefix LM



    - 数据组织
      - sentenceA with mask（[MASK]、[gMASK]） [sop] sentenceB [eop]

代码实战演练

ChatGLM

- **GLM和ChatGLM**
  - ChatGLM
    - 训练方式
      - v1 Prefix LM
      - v2 Causal LM
    - 数据格式
      - [Round N]\n\n问：Pormpt \n\n答：Response
    - 数据组织
      - v1 Prompt [gMASK] [sop] Response [eop]
      - v2 [gMASK] [sop] Prompt Response [eos]

代码实战演练

ChatGLM3

- **新版本特性**
  - 训练层面
    - 更多样的训练数据
    - 更充分的训练步数
    - 更合理的训练策略
  - 模型层面
    - 开放了base模型
  - 应用层面
    - 支持工具调用（Function Call）
    - 支持代码执行（Code Interpreter）

代码实战演练

ChatGLM3

- **工具调用**
  - 支持自定义工具进行调用

```
<|system|>
Answer the following questions as best as you can. You have access to the following tools:
[
    {
        "name": "get_current_weather",
        "description": "Get the current weather in a given location",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "The city and state, e.g. San Francisco, CA",
                },
                "unit": {"type": "string"},
            },
            "required": ["location"],
        },
    }
]
<|user|>
今天北京的天气怎么样？
<|assistant|>
好的，让我们来查看今天的天气
<|assistant|>get_current_weather
```python
tool_call(location="beijing", unit="celsius")
```
<|observation|>
{"temperature": 22}
<|assistant|>
根据查询结果，今天北京的气温为 22 摄氏度。
```

代码实战演练

ChatGLM3

- **代码执行**
  - 根据指令直接生成代码解决问题

````
<|user|>
帮我画一个爱心
<|assistant|>interpreter
```python
import numpy as np
import matplotlib.pyplot as plt

# 爱心形状的参数方程
def heart(t):
    x = 16 * np.sin(t) ** 3
    y = 13 * np.cos(t) - 5 * np.cos(2 * t) - 2 * np.cos(3 * t) - np.cos(4 * t)
    return x, y

t = np.linspace(0, 2 * np.pi, 1000)
x, y = heart(t)

plt.figure(figsize=(6, 6))
plt.plot(x, y, color='red')
plt.axis('equal')
plt.axis('off')
plt.show()
```
<|observation|>
```result
【image】
```
<|assistant|>
这是一个爱心形状。我使用了参数方程来描述这个形状，并使用matplotlib进行了绘制。如果您有任何其他需求或问题，请随时告诉我。
<|user|>  # End
````

代码实战演练

ChatGLM3

- **新版本特性**
  - 新版本Prompt设计
    - 整体样式

```
<|system|>
You are ChatGLM3, a large language model trained by Zhipu.AI. Follow the user's instructions carefully. Respond using markdown.
<|user|>
Hello
<|assistant|>
Hello, I'm ChatGLM3. What can I assist you today?
```

    - 对话头
      - <|role|>{metadata}
    - 角色表示
      - <|system|>,<|user|>,<|assistant|>,<|observation|>

代码实战演练（下，基于ChatGLM3）

- **数据集**
  - https://huggingface.co/datasets/shibing624/alpaca-zh
  - 指令微调
- **预训练模型**
  - ChatGLM3-6B-base
- **参数高效微调方法**
  - Lora

代码实战演练

模型训练细节

- **ChatGLM3 模型训练细节（调用chat进行对话）**
  - 数据格式要与chat方法的格式对齐，可通过查看chat方法对指令模板进行探索
  - special token需要单独处理，包括角色类（user、assistant等）和eos_token
  - 对于非工具调用功能和代码执行功能的训练，要求解码的第一个token一定是“\n”
  - 对于Lora配置中未明确task_type的情况，需要指定remove_unused_columns参数，否则会报奇怪的错误，很难debug

03

8bit模型训练实战

（1）量化简介

（2）相关环境配置

（3）代码实战演练

量化介绍

什么是量化

- 量化是一种模型压缩的方法，使用低精度数据类型对模型权重或者激活值进行表示。简单来说，量化就是将高精度的数字通过某种手段将其转换为低精度的数据
- 量化带来的优势
  - 降低模型加载的内存成本（FP32>FP16>INT8）
  - 减少运行计算成本，加速计算（并非所有情况都可以加速）
- 那么，如何进行量化?

量化介绍

- **INT8量化示例(absmax)**
  - 原始数据：x = [1.52, 2.64, -3.45, 4.32]
  - 量化过程：
    - x_absmax = 4.32
    - scale_factor = 127 / x_absmax ≈ 29.4
    - q_x = round([1.52, 2.64, -3.45, 4.32] * scale_factor) = [45, 78, -101, 127]
  - 反量化过程
    - x’ = q_x / scale_factor = [1.53, 2.61, 3.44, 4.32]

量化介绍

- **量化的问题**
  - 量化过程存在量化误差
    - 原始数据：x = [1.52, 2.64, -3.45, 4.32]
    - 反量化结果：x’ = [1.53, 2.61, 3.44, 4.32]
  - 如何降低量化误差，提高量化精度?
    - 使用更多的量化参数（scale_factor）
    - 矩阵乘法A*B可以看作是A的每一行乘上B的每一列，为A的每一行和B的每一列单独设置scale_factor，这种方式被称之为Vector-wise量化

量化介绍

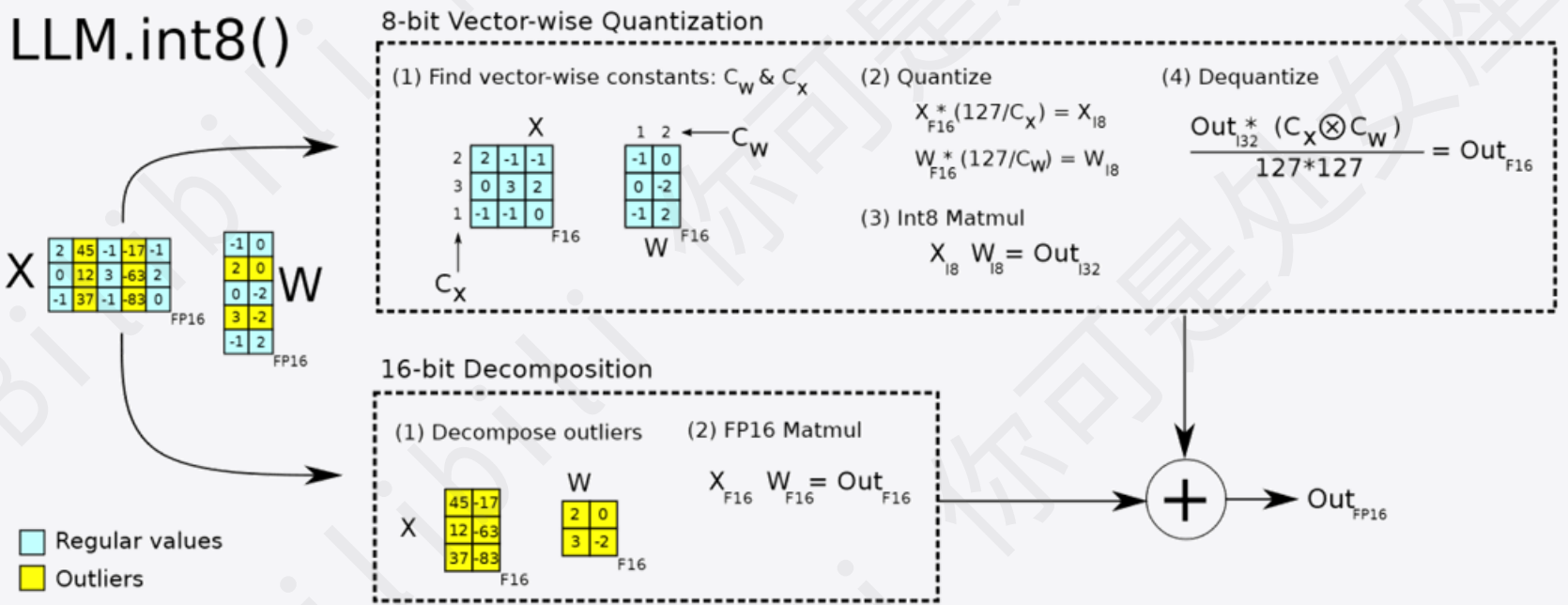
- **INT8量化示例2(absmax)**
  - 原始数据：x = [1.42, 1.51, 1.54, 45.3]
  - 量化过程：
    - x_absmax = 45.3
    - scale_factor = 127 / x_absmax ≈ 2.8
    - q_x = round([1.42, 1.51, 1.54, 4.32] * scale_factor) = [4, 4, 4, 127]
  - 反量化过程
    - x’ = q_x / scale_factor = [1.43, 1.43, 1.43, 45.36]

量化介绍

- **量化的问题**
  - 离群值：超出某个分布范围的值通常称为离群值
    - x = [1.42, 1.51, 1.54, **45.3**]
  - 8 位精度的动态范围极其有限，因此量化具有多个大值的向量会产生严重误差
  - 误差在一点点累积的过程中会导致模型的最终性能大幅度下降
  - 那么，如何解决这一问题？

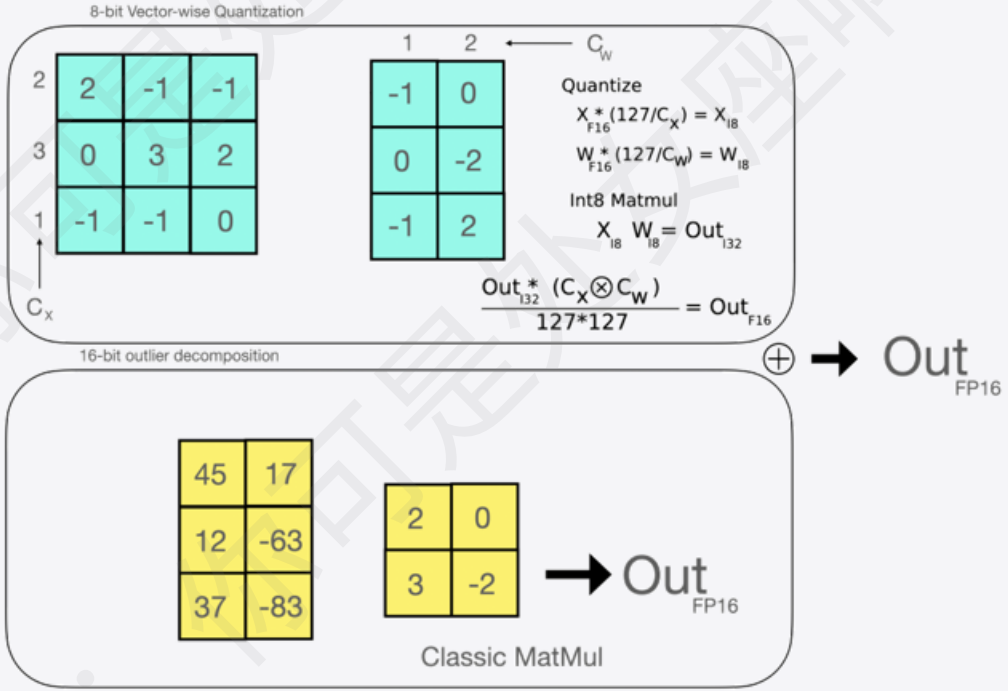
量化介绍

- **混合精度分解量化**



量化介绍

- **混合精度分解量化**
  - 从输入的隐含状态中，按列提取离群值 (即大于某个阈值的值)。
  - 对 FP16 离群值矩阵和 INT8 非离群值矩阵分别作矩阵乘法。
  - 反量化非离群值的矩阵乘结果并其与离群值矩阵乘结果相加，获得最终的 FP16 结果。



环境配置

- **环境配置**
  - peft 0.6.0
  - accelerate 0.23.0
  - transformers 4.34.0 (4.35.0 chatglm3截止2023.11.08有bug)
  - bitsandbytes 0.41.1 （torch+cu112以上）
    - https://github.com/jllllll/bitsandbytes-windows-webui/releases/tag/wheels
      - pip install bitsandbytes-0.41.1-py3-none-win_amd64.whl

代码实战演练

- **数据集**
  - https://huggingface.co/datasets/shibing624/alpaca-zh
  - 指令微调
- **预训练模型**
  - LLaMA2-7B
  - ChatGLM3-6B-base
- **参数高效微调方法**
  - Lora

04

4bit模型训练实战

（1）QLoRA介绍

（2）代码实战演练

QLoRA介绍

- **还是从量化说起**
  - 量化的本质
    - 将原本一个空间的数映射到另外一个空间
  - 回顾8bit量化
    - 将FP32映射到INT8，使用了简单的线性变换
    - (x / scale_factor) × 127 -> [-127，127]

QLoRA介绍

- **8bit 线性量化**
  - 原始数据：x = [1.55, 1.62, 1.83, 4.32]
  - 8bit 量化过程：
    - x_absmax = 4.32
    - scale_factor = 127 / x_absmax ≈ 29.4
    - q_x = round([1.52, 1.62, 1.83, 4.32] * scale_factor) = [46, 48, 54, 127]
  - 反量化：
    - x’ = [46, 48, 54, 127] / scale_factor = [1.56, 1.63, 1.84, 4.32]

QLoRA介绍

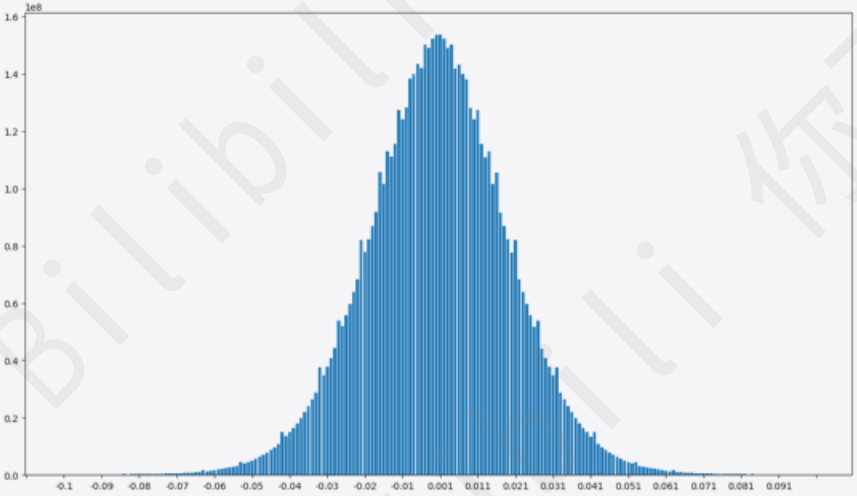
- **4bit 线性量化**
  - 原始数据：x = [1.55, 1.62, 1.83, 4.32]
  - 4bit 量化过程：
    - x_absmax = 4.32
    - scale_factor = 7 / x_absmax ≈ 1.62
    - q_x = round([1.52, 1.62, 1.83, 4.32] * scale_factor) = [3, 3, 3, 7]
  - 反量化：
    - x’ = [3, 3, 3, 7] / scale_factor = [1.85, 1.85, 1.85, 4.32]

QLoRA介绍

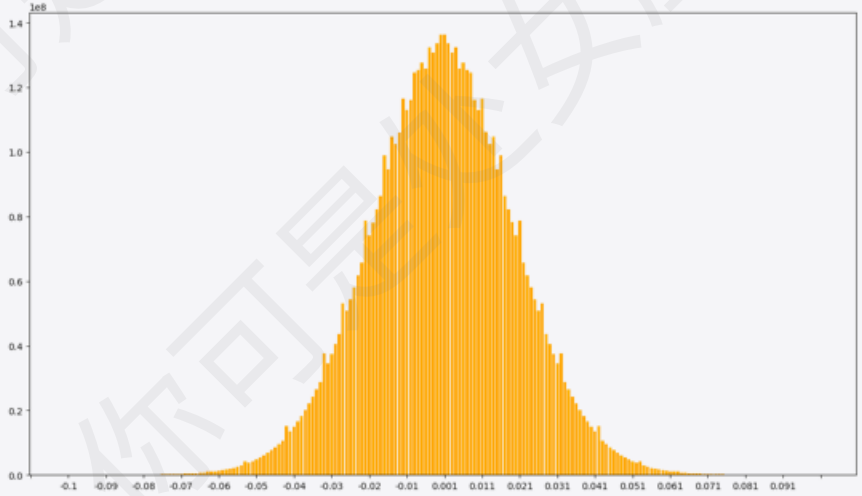
- **线性量化**
  - 简单线性量化的问题
    - 4 bit的表示范围比8bit更小，粒度更粗
    - 不用非常大的离群值，就使得量化后的参数都集中在某数值上
    - 量化误差会非常大
  - 如何理解线性量化
    - 数据归一化到[-1, 1]，把[-1, 1]均匀切分成N区间（N取决于量化bit）
    - 归一化的结果归属于第几个区间，量化结果便为几，数据的分布对量化结果非常重要
    - 如果待量化的数据为均匀分布，那么线性量化的结果即使是4bit也不会太差

QLoRA介绍

- **模型权重的真实分布**
  - 模型权重的真实分布



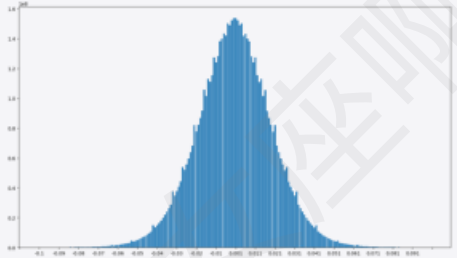
LLaMA2



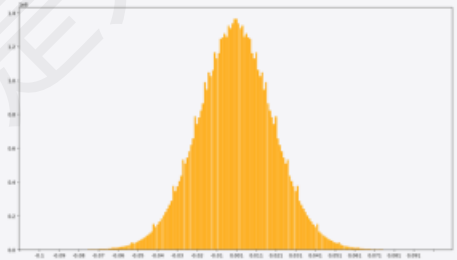
ChatGLM3

QLoRA介绍

- **正态分布**
  - 模型权重的真实分布
    - 权重的分布并非是均匀的
    - 权重的分布是呈现出一种正态分布的趋势的
  - 正态分布有什么特点
    - 中间多，两边少
    - 数值集中分布在均值两侧，离均值越远的部分越少



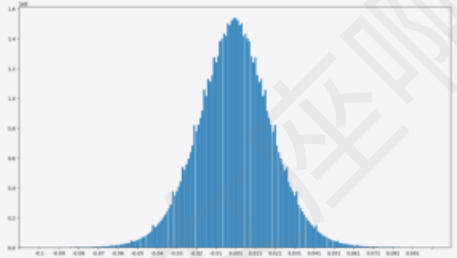
LLaMA2



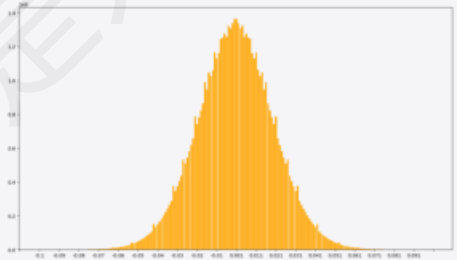
ChatGLM3

QLoRA介绍

- **如何与量化结合**
  - 更好的量化
    - 每部分的权重值均匀的落在N个区间中时量化效果会好
  - 线性量化的问题
    - 在目标域做均匀切分，无法将近似正态分布的值均匀的落在每个分区内
  - 换个思路
    - 目标域无法切分，那可以在源域上进行切分
    - 分位数量化



LLaMA2



ChatGLM3

QLoRA介绍

- **分位数量化**
  - 什么是分位数
    - 把顺序排列的一组数据分割为若干相等部分的分割点的数值即为相应的分位数
    - 中位数是分位数中最简单的一种，它将数据等分成两分。
    - 四分位数则是将数据按照大小顺序排序后，把数据分割成四等分的三个分割点上的数值。
  - 示例
    - {6, 7, 15, 36, 39, 40, 41, 42, 43, 47, 49}
    - 二分位数（中位数）：40
    - 四分位数：15，40，43

QLoRA介绍

- **分位数量化**
  - 分位数量化
    - 以4bit为例，表示范围为16个值，将权重从小到大排序，找到十五个分位数，将其切分为十六块，权重数值落在第几块，量化的表示就是多少，范围[0-15]
    - 此外，由于涉及到反量化，还需要给这16个值一个浮点数的映射，这个映射可以取分位区间两侧分位点的均值，用于反量化，这部分称之为量化值
    - 具体操作时，我们只需要把待量化的数跟量化值进行比较，取最相近的值即可作为该数值的量化值，对应的表示可以通过分位数进行确定，存储时同时存储4bit的表示与对应量化值，计算时使用量化值反量化后进行计算

QLoRA介绍

- **分位数量化改进-NF4**
  - 从正态分布入手
    - 大多数权重整体呈现正态分布，那么可以将其统一缩放至[-1，1]，根据标准正态分布得到16个量化值，并将量化值也缩放至[-1，1]，此时，便可利用前面提到的方法，将权重进行量化
    - 为了减少异常值的问题，采用分块量化，块大小为64，即64个值为一组进行量化
  - NF4
    - [-1.0, -0.6961928009986877, -0.5250730514526367, -0.39491748809814453, -0.28444138169288635, -0.18477343022823334, -0.09105003625154495, 0.0, 0.07958029955625534, 0.16093020141124725, 0.24611230194568634, 0.33791524171829224, 0.44070982933044434, 0.5626170039176941, 0.7229568362236023, 1.0]

QLoRA介绍

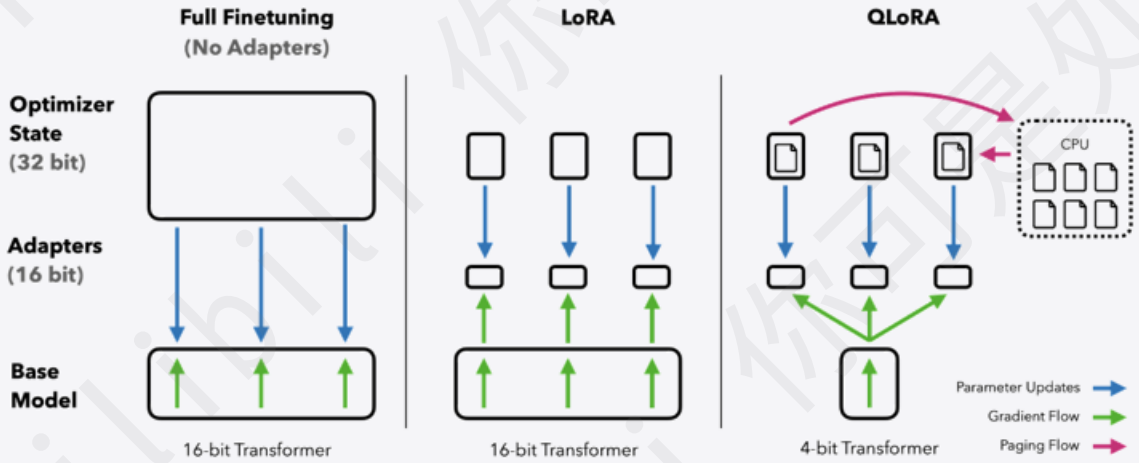
- **NF4 量化完整示例**
  - 原始数据：x = [1.55, 1.62, 1.83, 4.32]
  - 4bit 量化过程：
    - x_absmax = 4.32
    - x_norm = [1.55, 1.62, 1.67, 4.32] / 4.32 = [0.3587, 0.375, 0.4236, 1]
    - NF4 match：q_x = [0.3379, 0.3379, 0.4407, 1.0] = [11, 11, 12, 15]
  - 反量化过程：
    - x’ = [0.3379, 0.3379, 0.4407, 1.0] * x_absmax = [1.46, 1.46, 1.90, 4.32]

QLoRA介绍

- **双重量化**
  - NF4量化存储空间
    - NF4 分位点：16个常量值，可忽略不计
    - 量化常数（absmax）：每个块需要一个FP32的量化常数，32 / 64 = 0.5 bit，即每个参数除了4bit的值外，还会增加0.5bit的额外存储
  - 双重量化
    - 对量化常数进行二次量化，256个一组进行8bit量化，量化为fp8，
    - 8 / 64 + 32/(64 · 256) ≈ 0.127 bit
    - 相较于原始的0.5bit，每个参数额外的资源消耗减少了0.373bit

QLoRA介绍

- **分页优化器**
  - 分页优化器
    - 当显存不足时，将优化器的参数转移到CPU内存上，在需要时再将其取回，防止显存峰值时OOM



环境配置

- **环境配置**
  - peft 0.6.0
  - accelerate 0.23.0
  - transformers 4.34.0 (4.35.0 chatglm3截止2023.11.08有bug)
  - bitsandbytes 0.41.1
    - https://github.com/jllllll/bitsandbytes-windows-webui/releases/tag/wheels
      - pip install bitsandbytes-0.41.1-py3-none-win_amd64.whl

代码实战演练

- **数据集**
  - https://huggingface.co/datasets/shibing624/alpaca-zh
  - 指令微调
- **预训练模型**
  - LLaMA2-7B
  - ChatGLM3-6B-base
  - 13B / 20B / ...
- **参数高效微调方法**
  - Lora

低精度训练篇
完结

分布式训练篇

基于Accelerate的Transformers模型分布式训练解决方案

你可是处女座啊

目录

01 分布式训练基础与环境配置

（1）分布式训练简介

（2）如何进行分布式训练

（3）分布式训练环境配置

知识回顾

- **大模型训练背景介绍**
  - 计算效率
    - 大模型的训练依赖于海量的训练数据，海量的训练数据带来了海量的计算需求
  - 显存效率
    - 大模型主要体现在模型参数规模上，参数规模逐渐变大的模型对显存的依赖逐渐加剧

知识回顾

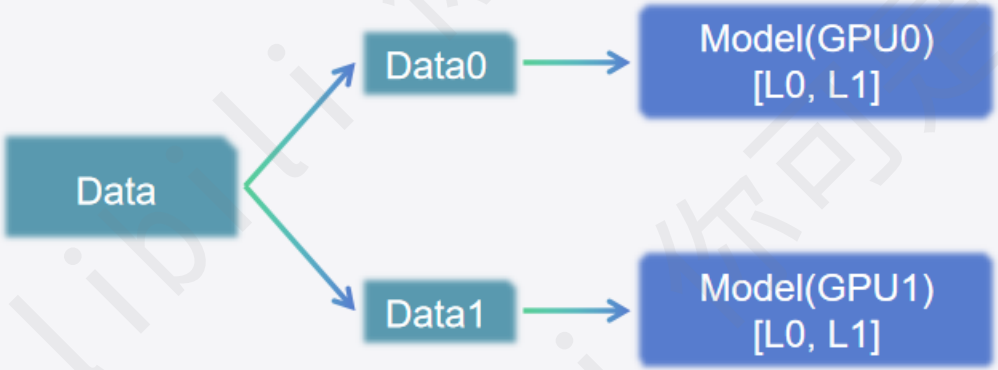
- **单卡场景如何解决显存问题**
  - 可训练参数量
    - 参数高效微调——PEFT
    - Prompt-Tuning、Prefix-Tuning、LoRA等
  - 参数精度
    - 低精度模型训练——Bitsandbytes
    - 半精度、INT8、NF4

分布式训练

- **分布式训练简介**
  - 什么是分布式模型训练
    - 分布式（Distributed）是指系统或计算任务被分布到多个独立的节点或计算资源上进行处理，而不是集中在单个节点或计算机上。
    - 分布式模型训练是一种机器学习和深度学习领域中的训练方法，它通过将计算任务分发到多个计算资源或设备上来加速模型的训练过程。分布式训练通过并行计算的方式，将数据和计算任务分配到多个节点上，从而提高训练速度和处理大规模数据的能力。

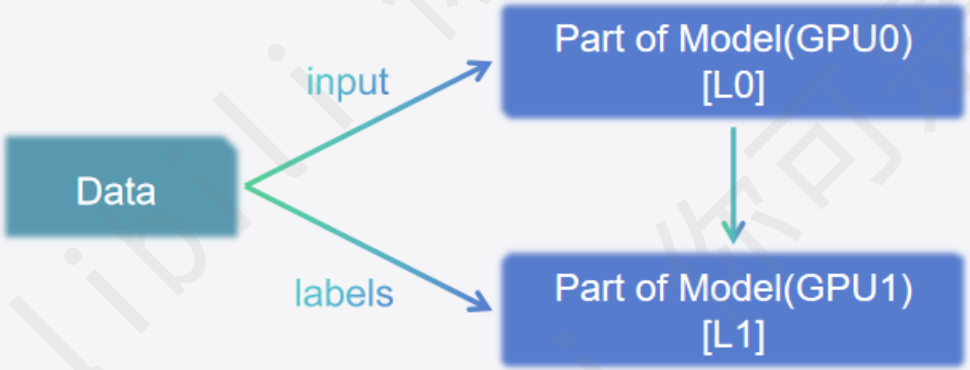
分布式训练

- **如何进行分布式模型训练**
  - 数据并行，Data Parrallel, DP
    - 每个GPU上都复制一份完整模型，但是每个GPU上训练的数据不同
    - 要求每张卡内都可以完整执行训练过程



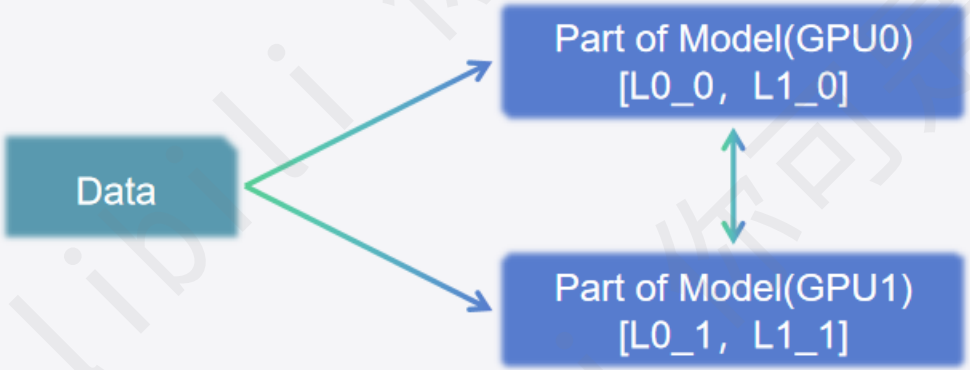
分布式训练

- **如何进行分布式模型训练**
  - 流水并行，Pipeline Parrallel, PP
    - 将模型按层拆开，每个GPU上包含部分的层，保证能够正常训练
    - 不要求每张卡内都可以完整执行训练过程



分布式训练

- **如何进行分布式模型训练**
  - 张量并行，Tensor Parrallel, TP
    - 将模型每层的权重拆开，对于一份权重，每个GPU上各包含一部分，保证能够正常训练
    - 不要求每张卡内都可以完整执行训练过程

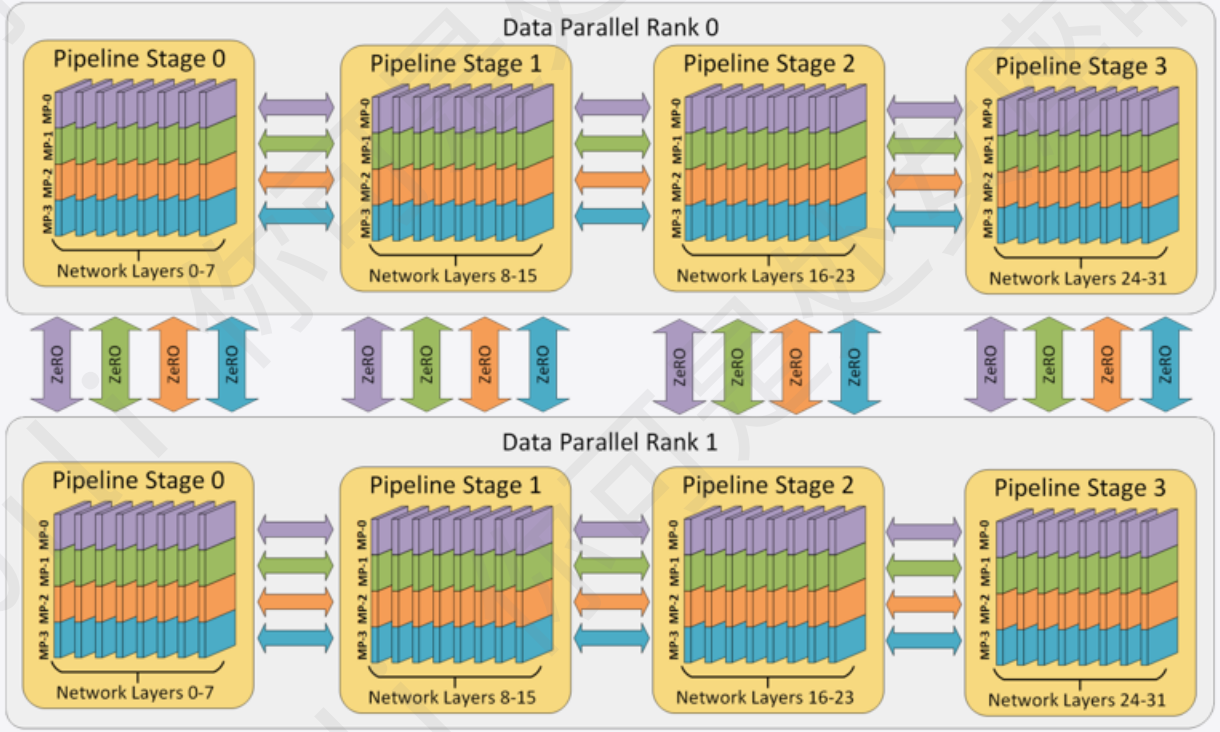


分布式训练

- **如何进行分布式模型训练**
  - 单卡可以完成训练流程的模型
    - 每个GPU上都复制一份完整模型，但是每个GPU上训练的数据不同（数据并行，Data Parrallel, DP）
  - 单卡无法完成训练流程的模型
    - 将模型按层拆开，每个GPU上包含部分的层，保证能够正常训练（流水并行，Pipeline Parrallel, PP）
    - 将模型每层的权重拆开，对于一份权重，每个GPU上各包含一部分，保证能够正常训练（张量并行，Tensor Parrallel, TP）
  - 混合策略
    - 数据并行 + 流水并行 + 张量并行（3D并行）

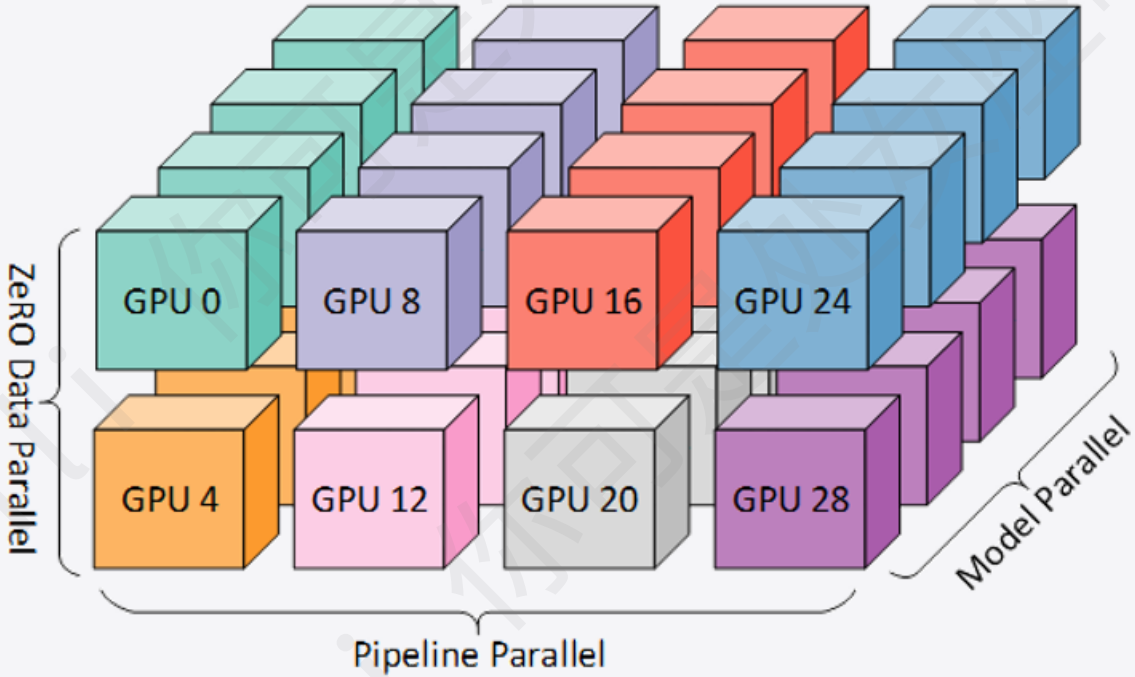
分布式训练

- **如何进行分布式模型训练**
  - 3D并行示例
    - 2路数据并行
    - 4路流水并行
    - 4路张量并行



分布式训练

- **如何进行分布式模型训练**
  - 3D并行示例
    - 2路数据并行
    - 4路流水并行
    - 4路张量并行



分布式训练环境配置

- **分布式训练环境配置**
  - 服务器显卡租赁
    - 趋动云、AutoDL
  - 环境安装
    - 设置 pypi 源 http://mirrors.aliyun.com/pypi/simple/
    - transformers==4.36.2 accelerate==0.26.1 evaluate datasets
  - VS Code 远程连接
    - remote ssh 远程连接
    - vscode python 插件安装

02

Data Parallel原理与应用

（1）Data Parallel 原理

（2）Data Parallel 训练实战

（3）Data Parallel 推理对比

Data Parrallel 原理

- **Data Parrallel 原理**
  - 什么是数据并行
    - 每个GPU里都存一份完整的模型，训练时每个GPU里的模型训练不同的数据
    - 适用于单卡能够运行完整训练流程的情况，如果单卡无法运行完整流程，则无法使用
  - Data Parrallel
    - 这里特指Pytorch框架中的nn.DataParallel所实现的数据并行方法

Data Parrallel 原理

- **Data Parrallel 原理**
  - 训练流程
    - Step1 GPU0 加载model和batch数据
    - Step2 将batch数据从GPU0均分至各卡
    - Step3 将model从GPU0复制到各卡
    - Step4 各卡同时进行前向传播
    - Step5 GPU0收集各卡上的输出，并计算Loss
    - Step6 将Loss分发至各卡，进行反向传播，计算梯度
    - Step7 GPU0收集各卡的梯度，进行汇总
    - Step8 GPU0更新模型

Data Parrallel 训练实战

- Data Parrallel 训练实战
  - nn.DataParallel源码实现

```
def forward(self, *inputs, **kwargs):
    with torch.autograd.profiler.record_function("DataParallel.forward"):
        if not self.device_ids:
            return self.module(*inputs, **kwargs)

        for t in chain(self.module.parameters(), self.module.buffers()):
            if t.device != self.src_device_obj:
                raise RuntimeError("module must have its parameters and buffers "
                                   "on device {} (device_ids[0]) but found one of "
                                   "them on device: {}".format(self.src_device_obj, t.device))

        inputs, kwargs = self.scatter(inputs, kwargs, self.device_ids)
        # for forward function without any inputs, empty list and dict will be created
        # so the module can be executed on one device which is the first one in device_ids
        if not inputs and not kwargs:
            inputs = ((),)
            kwargs = ({},)

        if len(self.device_ids) == 1:
            return self.module(*inputs[0], **kwargs[0])
        replicas = self.replicate(self.module, self.device_ids[:len(inputs)])
        outputs = self.parallel_apply(replicas, inputs, kwargs)
        return self.gather(outputs, self.output_device)
```

Data Parrallel 训练实战

- **Data Parrallel 训练实战**
  - 环境配置
    - 参考入门篇的环境安装，但是需要一台2张GPU及以上的训练机器
    - 如无合适机器，可以考虑使用GPU在线租赁平台：趋动云、AutoDL
  - 训练代码
    - 文本分类从零实现代码（入门篇第三节）

Data Parrallel 训练实战

- **Data Parrallel 训练实战**
  - 源码实现

```python
def _wrap_model(self, model, training=True, dataloader=None):
    if self.args.use_ipex:
        dtype = torch.bfloat16 if self.use_cpu_amp else torch.float32
        model = self.ipex_optimize_model(model, training, dtype=dtype)

    if is_sagemaker_mp_enabled():
        # Wrapping the base model twice in a DistributedModel will raise an error.
        if isinstance(self.model_wrapped, smp.model.DistributedModel):
            return self.model_wrapped
        return smp.DistributedModel(model, backward_passes_per_step=self.args.gradient_accumulation_steps)

    # train/eval could be run multiple-times - if already wrapped, don't re-wrap it again
    if unwrap_model(model) is not model:
        return model

    # Mixed precision training with apex (torch < 1.6)
    if self.use_apex and training:
        model, self.optimizer = amp.initialize(model, self.optimizer, opt_level=self.args.fp16_opt_level)

    # Multi-gpu training (should be after apex fp16 initialization) / 8bit models does not support DDP
    if self.args.n_gpu > 1 and not getattr(model, "is_loaded_in_8bit", False):
        model = nn.DataParallel(model)
```

Data Parrallel 训练实战

- **Data Parrallel 训练实战**
  - 环境配置
    - 参考入门篇的环境安装，但是需要一台2张GPU及以上的训练机器
    - 如无合适机器，可以考虑使用GPU在线租赁平台：趋动云、AutoDL
  - 训练代码
    - 文本分类从零实现代码（入门篇第三节）
    - 文本分类Trainer实现代码（入门篇第七节）

Data Parrallel

- **Data Parrallel 训练实战**
  - 实际效果
    - 调用了多GPU进行训练，但是训练速度没有多大，甚至有可能下降（可能大家会观测到不同的现象）
  - Data Parrallel 的问题
    - 单进程，多线程，由于GIL锁的问题，不能充分发挥多卡的优势
    - 由于Data Parrallel的训练策略问题，会存在一个主节点占用比其他节点高很多
    - 效率较低，每次训练开始都要重新同步模型，大模型的同步时间会较难接受
    - 只适用于单机训练，无法支持真正的分布式多节点训练

Data Parrallel

- **Data Parrallel 训练实战**
  - 实际效果
    - 调用了多GPU进行训练，但是训练速度没有多大，甚至有可能下降（可能大家会观测到不同的现象）
  - Data Parrallel 的问题
    - 单进程，多线程，由于GIL锁的问题，不能充分发挥多卡的优势
    - 由于Data Parrallel的训练策略问题，会存在一个主节点占用比其他节点高很多
    - 只适用于单机训练，无法支持真正的分布式多节点训练

nn.DataParrallel是目前不推荐的一种数据并行训练策略

Data Parrallel

- **Data Parrallel 训练实战**
  - DataParallel真的没有用吗
    - 并非如此
    - 对于并行推理，DataParallel可以派上用场！
  - DataParallel 并行推理验证
    - DataParallel.module.forward()
    - DataParallel.forward()
    - DataParallel.forward()改进版本

03

Distributed Data Parallel 原理与应用

（1）Distributed Data Parallel 并行原理

（2）分布式训练中的一些基本概念

（3）分布式训练中的通信基本概念

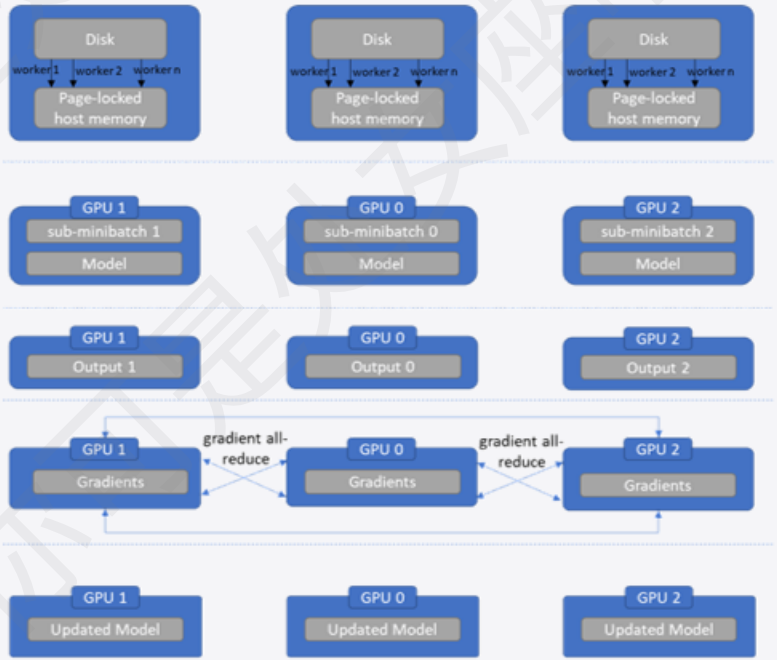
（4）Distributed Data Parallel 训练实战

Distributed Data Parrallel

- **为什么需要 Distributed Data Parrallel**
  - Data Parrallel 的问题
    - 单进程，多线程，由于GIL锁的问题，不能充分发挥多卡的优势
    - 由于Data Parrallel的训练策略问题，会存在一个主节点占用比其他节点高很多
    - 只适用于单机训练，无法支持真正的分布式多节点训练
    - nn.DataParrallel是目前不推荐的一种数据并行训练策略
  - 真正的分布式数据并行
    - Distributed Data Parrallel

Distributed Data Parallel 原理

- **Distributed Data Parallel 原理**
  - 训练流程
    - Step1 使用多个进程，每个进程都加载数据和模型
    - Step2 各进程同时进行前向传播，得到输出
    - Step3 各进程分别计算Loss，反向传播，计算梯度
    - Step4 各进程间通信，将梯度在各卡同步
    - Step5 各进程分别更新模型



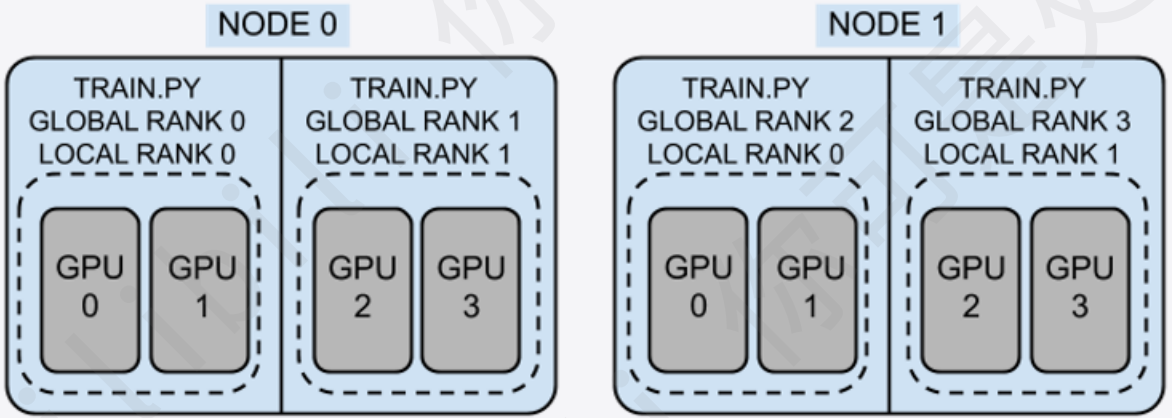
分布式训练中的通信

分布式训练中基本概念

- **分布式训练中的基本概念**
  - **group**：进程组，一个分布式任务对应一个进程组，一般就是所有卡都在一个组里
  - **world_size**：全局的并行数，一般情况下等于总的卡数
  - **node**：节点，可以是一台机器，或者一个容器，节点内包含多个GPU
  - **rank(global_rank)**：整个分布式训练任务内的进程序号
  - **local_rank**：每个node内部的相对进程序号

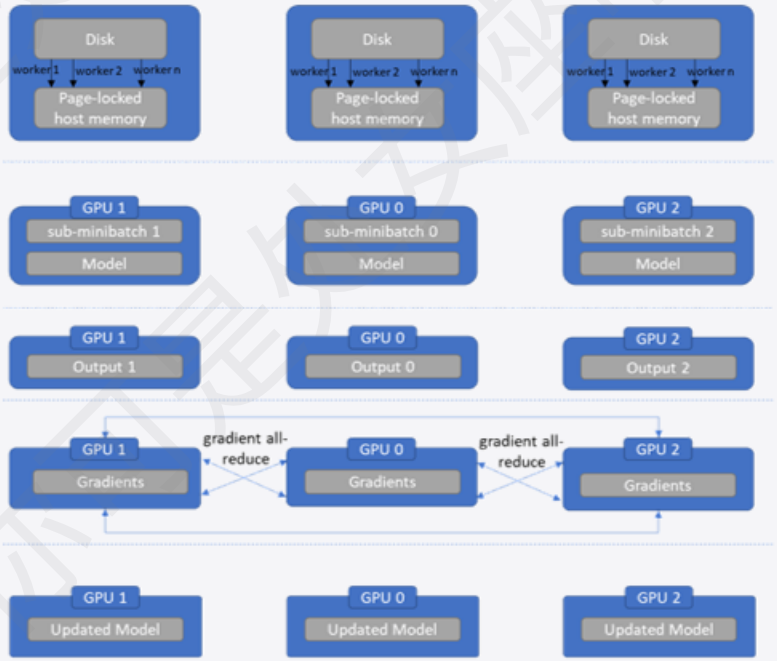
分布式训练中基本概念

- **分布式训练中的基本概念**
  - 2机4卡分布式训练示例
    - node=2, world_size=4,
    - 每个进程占用两个GPU



Distributed Data Parallel 原理

- **Distributed Data Parallel 原理**
  - 训练流程
    - Step1 使用多个进程，每个进程都加载数据和模型
    - Step2 各进程同时进行前向传播，得到输出
    - Step3 各进程分别计算Loss，反向传播，计算梯度
    - **Step4 各进程间通信，将梯度在各卡同步**
    - Step5 各进程分别更新模型

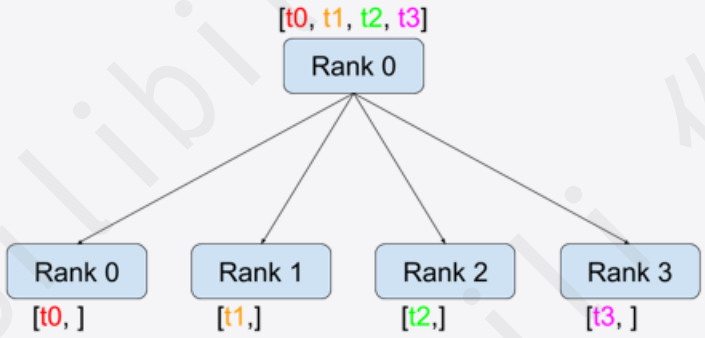


分布式训练中的通信

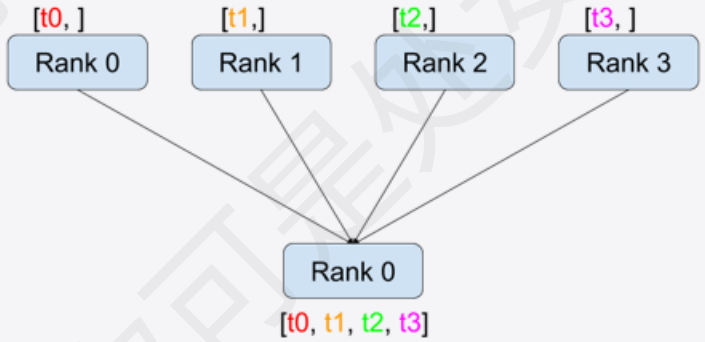
- **分布式训练中的通信**
  - 什么是通信
    - 在分布式模型训练中，通信是不同计算节点之间进行信息交换以协调训练任务的关键组成部分。
  - 通信类型
    - 点对点通信：将数据从一个进程传输到另一个进程称为点对点通信
    - 集合通信：一个分组中所有进程的通信模式称之为集合通信
      - 6种通信类型：Scatter、Gather、Reduce、All Reduce、Broadcast、All Gather

分布式训练中的通信

- **分布式训练中的通信**
  - 集合通信



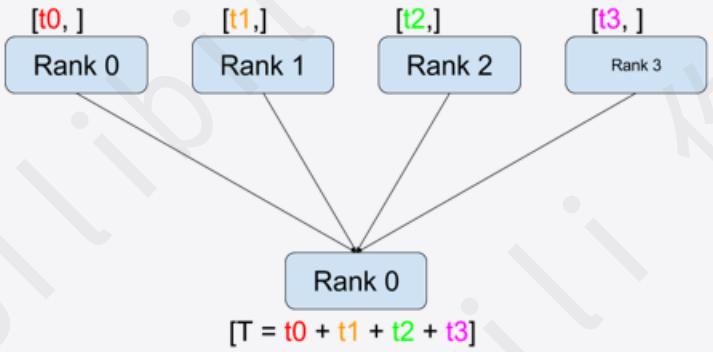
Scatter



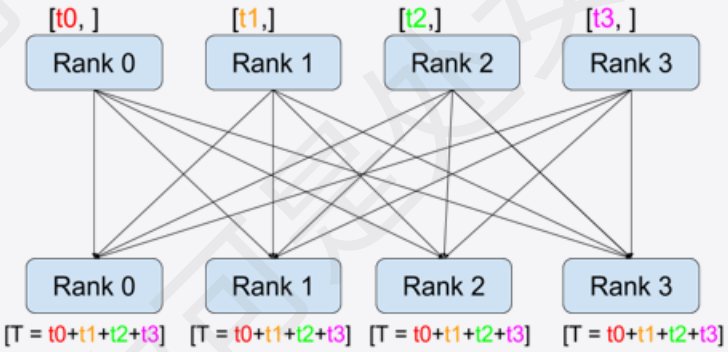
Gather

分布式训练中的通信

- **分布式训练中的通信**
  - 集合通信



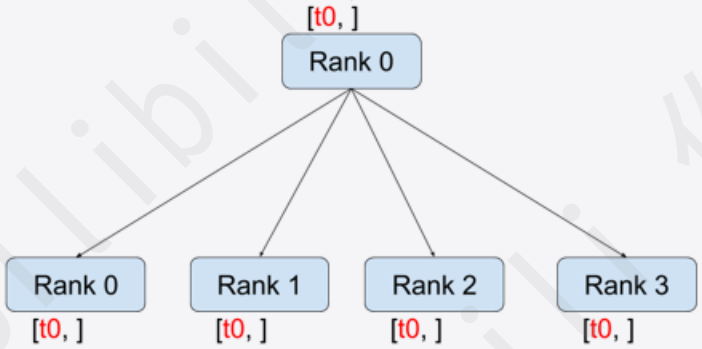
Reduce



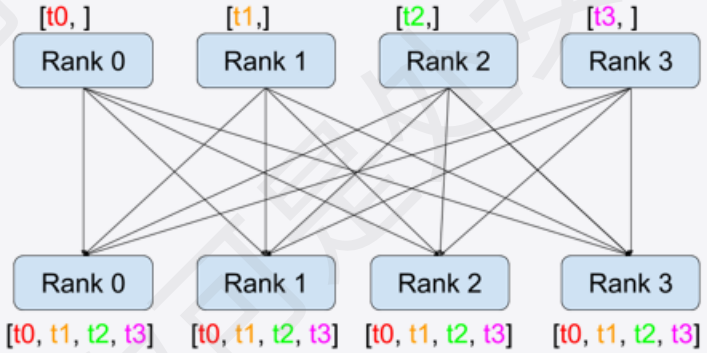
All Reduce

分布式训练中的通信

- **分布式训练中的通信**
  - 集合通信



Broadcast



All Gather

Distributed Data Parallel 训练实战

- **Distributed Data Parallel 训练实战**
  - 环境配置
    - 参考入门篇的环境安装，但是需要一台2张GPU及以上的训练机器
    - 如无合适机器，可以考虑使用GPU在线租赁平台：趋动云、AutoDL
  - 训练代码
    - 文本分类从零实现代码（入门篇第三节）
    - 文本分类Trainer实现代码（入门篇第七节）

Distributed Data Parallel 训练实战

- **DDP vs DP 效率对比**
  - chinese-roberta-wwm-base，单机2卡

Training Strategy	Num GPUs	Batch Size	Spend Time
None	1	64	97.0s
None	1	128	94.0s
None	1	256	OOM
DP	2	64	62.1s
DP	2	128	55.2s
DP	2	256	54.4s
DDP	2	64	63.8s
DDP	2	128	53.6s
DDP	2	256	50.7s

Distributed Data Parallel 训练实战

- **DDP vs DP 效率对比**
  - chinese-roberta-wwm-large，单机2卡

Training Strategy	Num GPUs	Batch Size	Spend Time
None	1	64	331.3s
None	1	128	OOM
None	1	256	OOM
DP	2	64	228.3s
DP	2	128	187.8s
DP	2	256	OOM
DDP	2	64	179.3s
DDP	2	128	165.6s
DDP	2	256	OOM

Distributed Data Parallel 训练实战

- **Distributed Data Parallel 实现注意细节**
  - 数据部分
    - 如果是在训练进程内对数据集进行划分，注意保证数据划分的一致性，可以通过随机种子控制
    - 分布式采样器会为了保证每个进程内数据大小一致，做额外的填充，评估指标可能会存在误差
  - 书写逻辑
    - 将分布式的代码看作单进程的代码即可，只是需要分布式的数据采样器以及启动略有不同
    - print打印的都是各自进程内的信息，需要全局的信息则需要自行调用通信计算结果
    - 数据放置到指定设备上时需要注意使用正确的device_id，一般用local_rank

04 Accelerate 基础入门

（1）Accelerate 基本介绍

（2）基于Accelerate DDP代码实现

（3）Accelerate 启动命令介绍

Accelerate 基础入门

- **Accelerate 简介**
  - 什么是Accelerate
    - Accelerate是Huggingface生态中针对分布式训练推理提供的库，目标是简化分布式训练的流程
    - Accelerate库本身不提供分布式训练的内容，但是其内部集成了多种分布式训练框架
      - DDP、FSDP、Deepspeed等
    - Accelerate库提供了统一的接口，一套代码搞定多种分布式训练框架，简单的几行代码（4行），便可让单机训练的程序变为分布式训练程序
    - Transformers库中也是通过Accelerate集成的分布式训练，因此Accelerate库的学习非常有必要！

Accelerate 基础入门

- **Accelerate 简介**

```
+ from accelerate import Accelerator
+ accelerator = Accelerator()

+ model, optimizer, training_dataloader, scheduler = accelerator.prepare(
+     model, optimizer, training_dataloader, scheduler
+ )

  for batch in training_dataloader:
      optimizer.zero_grad()
      inputs, targets = batch
      inputs = inputs.to(device)
      targets = targets.to(device)
      outputs = model(inputs)
      loss = loss_function(outputs, targets)
+     accelerator.backward(loss)
      optimizer.step()
      scheduler.step()
```

Accelerate DDP 训练实战

- **Accelerate DDP 训练实战**
  - 环境配置
    - 参考入门篇的环境安装，但是需要一台2张GPU及以上的训练机器
    - 如无合适机器，可以考虑使用GPU在线租赁平台：趋动云、AutoDL
  - 训练代码
    - DDP 版本的训练代码

Accelerate 基础入门

- **Accelerate 启动命令**
  - 创建配置文件
    - accelerate config
  - 启动命令
    - accelerate launch {script.py}
  - 查看帮助
    - accelerate launch --help

05

Accelerate 使用进阶

（1）Accelerate 混合精度训练

（2）Accelerate 梯度累积功能

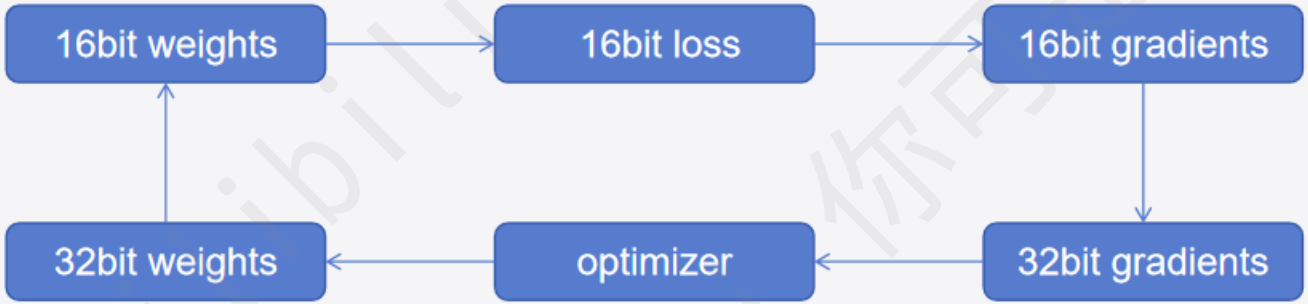
（3）Accelerate 实验记录功能

（4）Accelerate 模型保存功能

（5）Accelerate 断点续训功能

Accelerate 使用进阶

- **混合精度精训练**
  - 什么是混合精度训练
    - 混合精度训练是一种提高神经网络训练效率的技术，它结合了32位的单精度（FP32）浮点数和16位的半精度（FP16/BF16）浮点数来进行模型的训练。混合精度训的方法可以减少GPU内存的使用，同时加速训练过程。



Accelerate 使用进阶

- **混合精度精训练**
  - 混合精度训练中的显存占用
    - 假设模型参数量为M

	混合精度训练	单精度训练
模型	（4+2）Bytes * M	4 Bytes * M
优化器	8 Bytes * M	8 Bytes * M
梯度	（2 + ） Bytes * M	4 Bytes * M
汇总	(16 +) Bytes * M	16 Bytes * M

混合精度训练可以加速训练，但并不一定会降低显存占用！

Accelerate 使用进阶

- **混合精度精训练**
  - 混合精度训练中的显存占用
    - 假设模型参数量为M

	混合精度训练	单精度训练
模型	（4+2）Bytes * M	4 Bytes * M
优化器	8 Bytes * M	8 Bytes * M
梯度	（2 + ）Bytes * M	4 Bytes * M
激活值	2 Bytes * A	4 Bytes * A
汇总	(16 +) Bytes * M + 2 Bytes * A	16 Bytes * M + 4 Bytes * A

激活值占比较大时，可以看到明显的显存占用降低！

Accelerate 使用进阶

- **Accelerate 混合精度训练**
  - 方式一
    - accelerator = Accelerator(mixed_precision="bf16")
  - 方式二
    - acclerator config && choice bf16
  - 方式三
    - accelerator launch --mixed_precision bf16 {script.py}

Accelerate 使用进阶

- **梯度积累功能**
  - 什么是梯度累积
    - 梯度累积是一种深度学习训练技术，它允许模型在有限的硬件资源下模拟更大批量大小的训练效果。
    - 梯度累积的具体做法
      - 分割Batch：将大的训练数据Batch分割成多个小的Mini-Batch。
      - 计算梯度：对每个Mini-Batch独立进行前向和反向传播，计算梯度。
      - 累积梯度：不立即更新模型参数，而是将这些小Batch的梯度累积起来。
      - 更新参数：当累积到一定数量的梯度后，再统一使用这些累积的梯度来更新模型参数

Accelerate 使用进阶

- **梯度积累功能**
  - 梯度累积实现

```
accumulation_steps = 4  # 设定累积步数
model.zero_grad()       # 清空梯度
for step, (inputs, targets) in enumerate(data_loader):
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss = loss / accumulation_steps  # 缩放损失
    loss.backward()                   # 计算梯度
    if (step + 1) % accumulation_steps == 0:
        optimizer.step()              # 更新参数
        model.zero_grad()             # 清空梯度
```

Accelerate 使用进阶

- **Accelerate 梯度累积功能**
  - 步骤一
    - 创建Accelerator时指定梯度积累的大小
    - accelerator = Accelerator(gradient_accumulation_steps=xx)
  - 步骤二
    - 训练过程中，加入accelerator.accumulate(model)的上下文
    - with accelerator.accumulate(model):

Accelerate 使用进阶

- **实验记录功能**
  - 常见的实验记录工具
    - Tensorboard
    - WandB
    - CometML
    - Aim
    - MLflow
    - Neptune
    - Visdom

Accelerate 使用进阶

- **Accelerate 日志记录功能**
  - 步骤一
    - 创建Accelerator时指定project_dir
    - accelerator = Accelerator(log_with="tensorboard", project_dir="xx")
  - 步骤二
    - 始化tracker
    - accelerator.init_trackers(project_name="xx")
  - 步骤三
    - 结束训练，确保所有tracker结束
    - accelerator.end_training()

Accelerate 使用进阶

- **模型保存功能**
  - 模型保存内容
    - 模型权重，pytorch_model.bin / model.safetensors
    - 模型配置文件，关于模型结构描述的信息，一般是config.json
    - 其他文件，generation_config.json、adapter_model.safetensors
  - 如何进行模型保存
    - 单机训练的情况，调用model.save_pretrained(save_directory)即可
    - 分布式训练情况
      - 直接调用model.save_pretrained(save_directory)会报错，需要去包装
      - 并非所有进程都需要存，主进程保存即可

Accelerate 使用进阶

- **Accelerate 模型保存功能**
  - 方式一
    - accelerator.save_model()

Accelerate 使用进阶

- **Accelerate 模型保存功能**
  - 方式一
    - accelerator.save_model()
    - 存在问题
      - 不保存配置文件，只保存模型参数
      - 对于PEFT的模型支持不好，会将完整模型保存

Accelerate 使用进阶

- **Accelerate 模型保存功能**
  - 方式一
    - accelerate.save(save_directory)
  - 方式二
    - accelerator.unwrap_model(model).save_pretrained()
    - 指除了save_directory外，还需要指定is_main_process、state_dict、save_func参数

Accelerate 使用进阶

- **断点续训功能**
  - 什么是断点续训
    - 当训练过程因为某些原因（如停电、系统崩溃等）被中断时，断点续训允许我们从上次中断的地方恢复训练，而不是从头开始。这样可以节省大量的时间和计算资源。
  - 如何进行断点续训
    - 保存检查点（checkpoint）
    - 加载检查点（模型权重、优化器状态、学习率调度器、随机状态）
    - 跳过已训练数据（epoch、batch）

Accelerate 使用进阶

- **Accelerate 断点续训功能**
  - 步骤一
    - 保存检查点
    - accelerator.save_state()
  - 步骤二
    - 加载检查点
    - accelerator.load_state()
  - 步骤三
    - 计算跳过的轮数和步数
    - resume_epoch、resume_step
  - 步骤四
    - 数据集跳过对应步数
    - accelerator.skip_first_batches(trainloader, resume_step)

06

Accelerate + Deepspeed

（1）Deepspeed 介绍

（2）Accelerate + Deepspeed 代码实战

DDP 背景回顾

- **DDP 背景回顾**
  - 数据并行
    - 每个GPU里都存一份完整的模型，训练时每个GPU里的模型训练不同的数据
  - DDP 存在的问题
    - 单卡至少需要16 * M * Bytes的资源，M为模型参数量
    - 对于大模型全量参数微调，这基本是一件不可能的事情

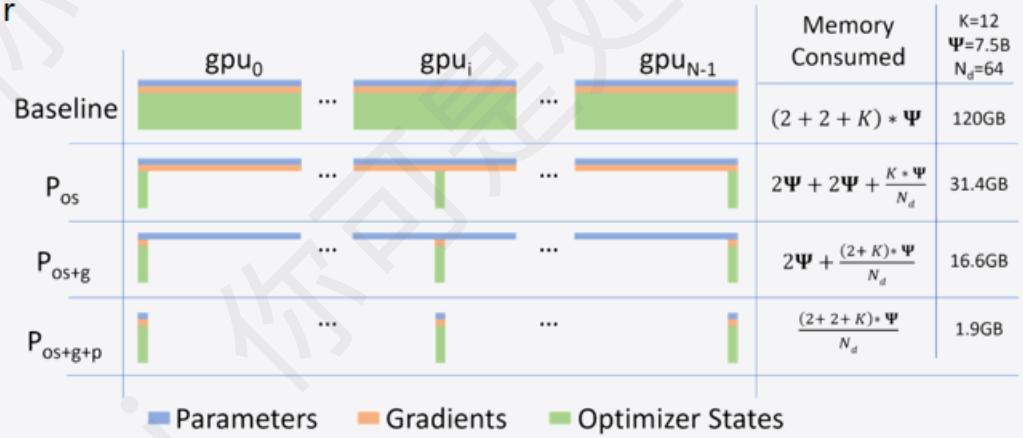
DDP 背景回顾

- **DDP 背景回顾**
  - 数据并行
    - 每个GPU里都存一份完整的模型，训练时每个GPU里的模型训练不同的数据
  - DDP 存在的问题
    - 单卡至少需要16 * M * Bytes的资源，M为模型参数量
    - 对于大模型全量参数微调，这基本是一件不可能的事情

存在冗余，N张卡上进行DDP训练，内存中需要加载N份模型，N份梯度，N份优化器

Deepspeed 介绍

- **Deepspeed 介绍**
  - Deepspeed介绍
    - DeepSpeed 是一个由微软开发的深度学习优化库，目标是使分布式训练变得简单、高效、有效。
    - ZeRO, Zero Redundancy Optimizer
      - ZeRO1, Optimizer States
      - ZeRO2, OS+Gradient
      - ZeRO3, OS+G+Parameter



Deepspeed 介绍

- **Deepspeed 介绍**
  - ZeRO策略通信量分析
    - ZeRO1, Optimizer States
      - 通信与优化器状态无关，因此与DDP相比通信总量不变
    - ZeRO2, Optimizer States+Gradient
      - 每个设备仅需要部分梯度，因此与DDP相比通信总量仍然不变
    - ZeRO3, Optimizer States+Gradient+Parameter
      - 由于需要额外的参数通信，与DDP相比通信总量提升1.5倍

Deepspeed 介绍

Deepspeed 介绍

- **Deepspeed 介绍**
  - Deepspeed介绍
    - ZeRO ++
      - 针对ZeRO3通信进行优化，通过权重量化、权重分层存储、梯度量化降低跨节点通信量
    - ZeRO offload
      - 将优化器、模型参数从显存卸载至内存，或者二者协同搭配（offload ++）
    - DeepSpeed Ulysses
      - 针对长序列训练，将各个样本在序列维度上分割给参与的GPU

Accelerate Deepspeed 集成

- **Accelerate Deepspeed集成**
  - Accelerate Deepspeed集成 方式一
    - Step1 配置Deepspeed相关信息
      - accelerate config
    - Step2 使用Accelerate命名启动脚本
      - accelerate launch ddp_accelerate.py

Accelerate Deepspeed 集成

- **Accelerate Deepspeed集成**
  - Accelerate Deepspeed集成 方式二
    - Step1 配置Deepspeed相关信息
      - accelerate config
    - Step2 配置完整Deepspeed Config
      - Deepspeed官网Config介绍 / Accelerate Deepspeed页面介绍
    - Step3 使用Accelerate命名启动脚本
      - accelerate launch ddp_accelerate.py

Accelerate Deepspeed 集成

Accelerate Deepspeed 集成

- **Accelerate Deepspeed集成**
  - 多机多卡
    - 直连情况下，指定deepspeed_hostfile, num_machines, num_processes, main_process_port，正常通过Accelerate启动
    - slurm管理情况下，配置启动器为torchrun标准的启动器，指定num_machines, machine_rank, num_processes, main_process_port, main_process_ip，还是以Accelerate启动，模式与Torchrun启动类似，或者直接使用Torchrun启动

分布式训练篇
完结