# tensorrt+triton模型部署实践

## 基础概念

### ONNX

ONNX(Open Neural Network Exchange)，开放神经网络交换，是一种模型IR，用于在各种深度学习训练和推理框架转换的一个中间表示格式。在实际业务中，可以使用Pytorch或者TensorFlow训练模型，导出成ONNX格式，然后在转换成目标设备上支撑的模型格式，比如TensorRT Engine、NCNN、MNN等格式。ONNX定义了一组和环境，平台均无关的标准格式，来增强各种AI模型的可交互性，开放性较强。Pytorch、tensorflow、Scikit-learn模型可以转化为ONNX格式。

Onnx的使用：

|  |
| --- |
| import onnx  import torch  from onnxruntime.quantization import QuantType, quantize\_dynamic  from onnxruntime.transformers import optimizer  from onnxruntime.transformers.float16 import convert\_float\_to\_float16  from onnxruntime.transformers.fusion\_options import FusionOptions  from onnxruntime.transformers.fusion\_utils import FusionUtils  from onnxruntime.transformers.onnx\_model import OnnxModel  from onnxruntime.transformers.onnx\_model\_bert import BertOnnxModel  from onnxruntime.transformers.optimizer import MODEL\_TYPES  def verify\_onnx\_model(model\_path):      onnx.load(model\_path)      onnx.checker.check\_model(model)  def get\_onnx\_input\_output(model\_name='bert-product-ner-opt.onnx'):      onnx\_model = onnx.load(model\_name)      print(onnx\_model.graph.input)      print(onnx\_model.graph.output)    def to\_onnx(model):      batch\_size = 32      inputs = (torch.randint(0, 21128, size=(batch\_size, 512)).long(),                torch.zeros(size=(batch\_size, 512)).long(),                torch.ones(size=(batch\_size, 512)).long())      onnx\_export = 'bert-product-ner.onnx'      input\_names = ['input\_ids', 'token\_type\_ids', 'attention\_mask']      output\_names = ["output1"]      model.eval()      torch.onnx.export(model, inputs, onnx\_export,                        export\_params=True,                        verbose=False,                        opset\_version=13,                        do\_constant\_folding=True,                        input\_names=input\_names,                        output\_names=output\_names,  # output names                        dynamic\_axes={'input\_ids' : {0 : 'batch\_size'},                                      'token\_type\_ids' : {0 : 'batch\_size'},                                      'attention\_mask' : {0 : 'batch\_size'},                                      'output1' : {0 : 'batch\_size'}                                      },                        training=torch.onnx.TrainingMode.EVAL                        )      verify\_onnx\_model(onnx\_export)  get\_onnx\_input\_output()  def optimize\_onnx(      onnx\_path: str,      onnx\_optim\_model\_path: str,      fp16: bool,      use\_cuda: bool,      num\_attention\_heads: int = 0,      hidden\_size: int = 0,      architecture: str = "bert"):        optimization\_options = FusionOptions(model\_type=architecture)      optimization\_options.enable\_gelu\_approximation = False      if architecture == "distilbert":          optimization\_options.enable\_embed\_layer\_norm = False      if architecture not in MODEL\_TYPES:          logging.info(f"Unknown architecture {architecture} for Onnx Runtime optimizer, overriding with 'bert' value")          architecture = "bert"      opt\_level = 1 if architecture == "bert" else 0      optimized\_model: BertOnnxModel = optimizer.optimize\_model(          input=onnx\_path,          model\_type=architecture,          use\_gpu=use\_cuda,          opt\_level=opt\_level,          num\_heads=num\_attention\_heads,  # automatic detection with 0 may not work with opset 13 or distilbert models          hidden\_size=hidden\_size,  # automatic detection with 0          optimization\_options=optimization\_options,      )      if fp16:          # use\_symbolic\_shape\_infer set to false because doesn't work after ONNX package v1.10.2          optimized\_model.convert\_float\_to\_float16(use\_symbolic\_shape\_infer=False)  # FP32 -> FP16      logging.info(f"optimizations applied: {optimized\_model.get\_fused\_operator\_statistics()}")      optimized\_model.save\_model\_to\_file(onnx\_optim\_model\_path) |

### **TensorRT Engine**

TensorRT是一个有助于在NVIDIA图形处理单元（GPU）上高性能推理c++库。专门致力于在GPU上快速有效地进行网络推理网络架构。由C++、CUDA、python等语言开发，使用灵活高效。

特性：

1.层与张量融合，比如网络向前传播时候CNN、Pooling、ReLU融合成一个算子。

1. 动态张量显存，最大限度减少显存得占用
2. 权重与激活精度校准，可以使用fp16,int8, int4精度对模型进行量化
3. 内核自动调整，据不同的显卡构架选择不同的优化策略以及计算方式，寻找最合适当前构架的计算方式。
4. 并行处理，多输入流并行处理。

**TensorRT的安装使用：**

拉取英伟达镜像

docker pull nvcr.io/nvidia/tensorrt:22.07-py3

运行docker容器

sudo docker run --gpus all -itd --rm -v $PWD:/workspace --name tensorrt nvcr.io/nvidia/tensorrt:22.07-py3

执行tensorrt命令

trtexec --onnx=bert-product-ner.onnx \

--minShapes="input\_ids:1x512, token\_type\_ids:1x512, attention\_mask:1x512" \

--optShapes="input\_ids:32x512, token\_type\_ids:32x512, attention\_mask:32x512" \

--maxShapes="input\_ids:32x512, token\_type\_ids:32x512, attention\_mask:32x512" \

--saveEngine=model.trt --fp16 --workspace=1024

--onnx指定onnx模型路径

--minShapes 动态输入的最小形状

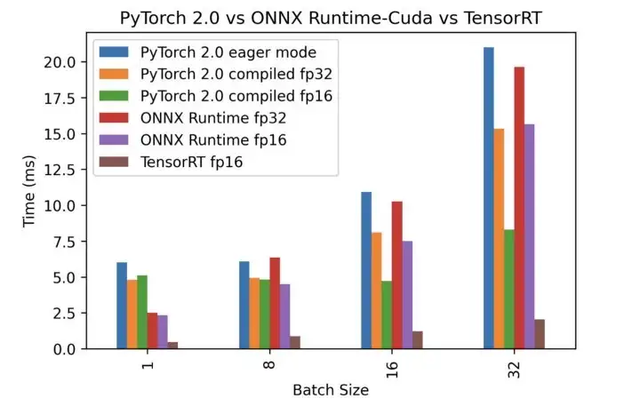
--optShapes 优化的输入的形状

--maxShapes 最大的输入形状

--saveEngine 保存文件的路径

--fp16 使用fp16量化裁剪模型

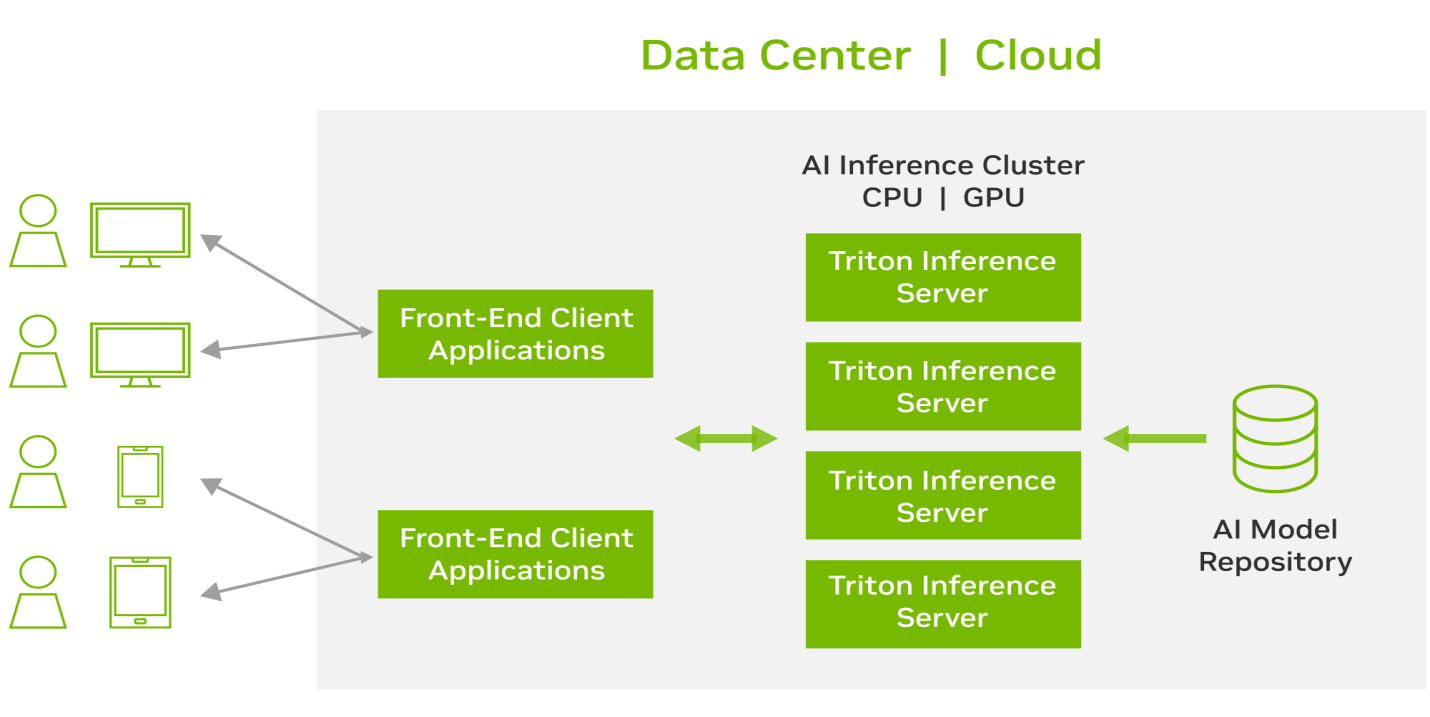
--workspace tensorrt运行占用的内存MB



tensorrt和优化后的pytorch静态图推理速度相比通常会快5到10倍。

### **1.3 Triton inference server**

NVIDIA Triton inference server是一款开源推理服务软件，可标准化 AI 模型部署和执行，并在生产中提供快速且可扩展的 AI, 能够在任何基于 GPU 或 CPU 的基础设施上的任何框架中部署、运行和扩展经过训练的 ML 或 DL 模型，从而简化和标准化 AI 推理。



1. 多平台支持，支持Pytorch、tensorflow、python、onnx、tensorrt、c++等模型格式,包括传统机器学习模型和深度学习模型以及大模型。
2. 高性能推理，支持动态批次，高并发请求，多模型集成等特性
3. 容易维护和扩展，集成k8s，导出监控指标监控服务的性
4. 支持LLM模型高性能部署，采用FastTransformer模块加速模型。

Triton安装使用

docker pull nvcr.io/nvidia/tritonserver:22.07-py3

model\_name='bert-product-ner'

sudo docker run -it --rm --gpus device=1 \

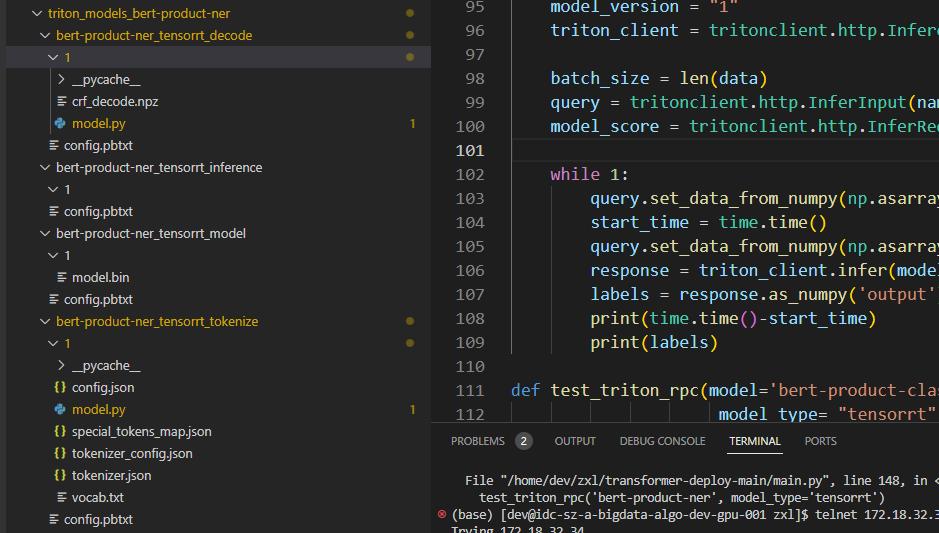
-p8003:8000 -p8004:8001 -p8005:8002 --shm-size 256m \

--name tritonserver \

-v $PWD/triton\_models\_${model\_name}:/models zuoxiaolei/tritonserver:22.07-py3 \

bash -c "tritonserver --model-repository=/models"

## **Triton部署tensorrt模型**

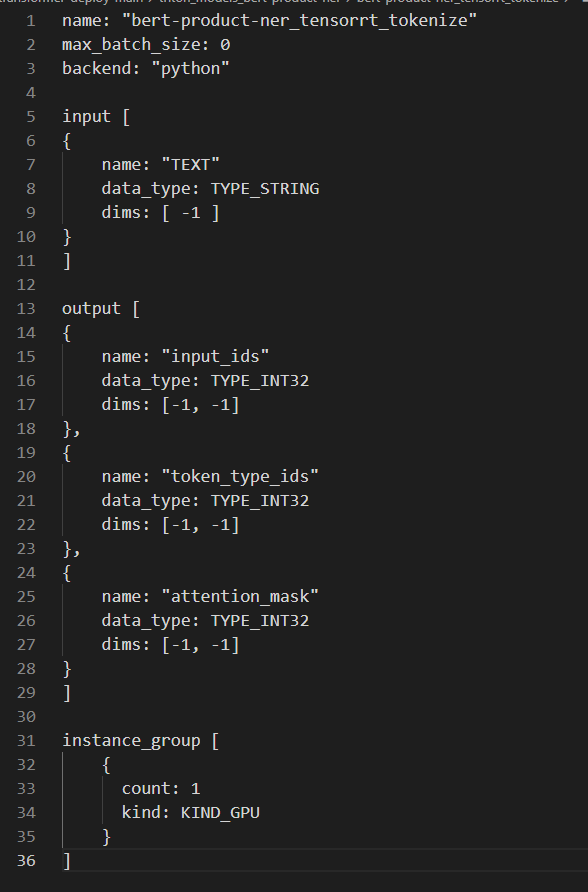


部署模型的文件结构，最上层目录为模型总的目录，存放模型pipeline各环境推理的任务。子目录存放各个子任务的模型配置文件和模型文件

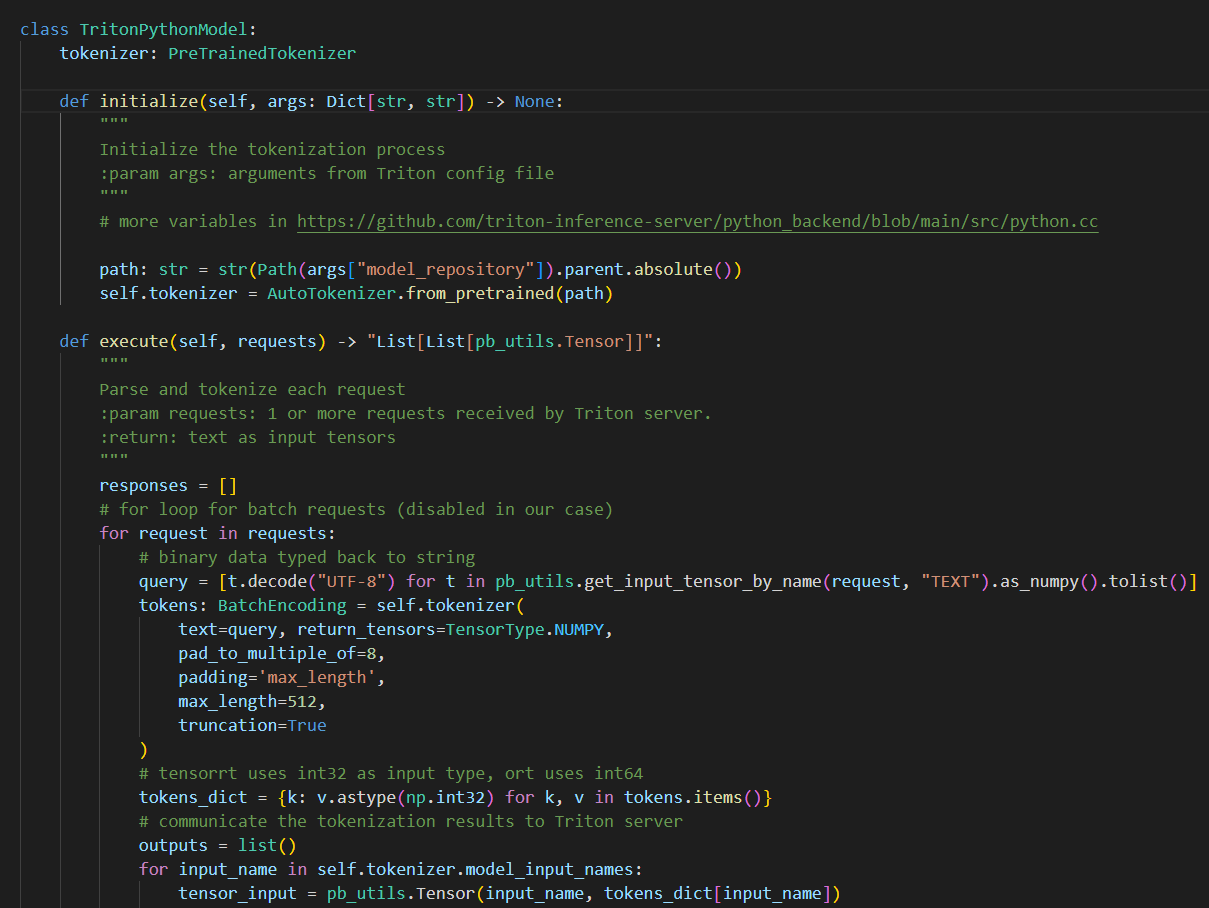
* 1. tokenize任务的创建。将模型转化为huggingface格式， 保存tokenizer文件后续放入triton容器中bert-product-ner\_tensorrt\_tokenize使用。

|  |
| --- |
| def to\_huggingface\_model():      tokenizer = tokenizer\_class.from\_pretrained(args.output\_dir, do\_lower\_case=args.do\_lower\_case)      config = config\_class.from\_pretrained(args.model\_name\_or\_path, num\_labels=num\_labels)      model = model\_class.from\_pretrained(args.model\_name\_or\_path, config=config)      model.eval()      BertCrfForNer.register\_for\_auto\_class("AutoModel")      BertCrfForNer.register\_for\_auto\_class("AutoModelForTokenClassification")      output\_directory = "../models/bert-product-ner"      model.save\_pretrained(output\_directory)      tokenizer.save\_pretrained(output\_directory) |

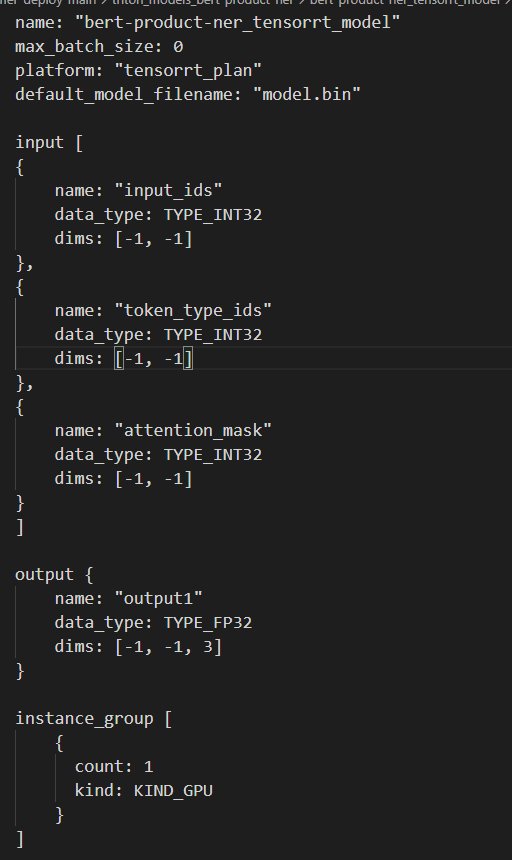
修改bert-product-ner\_tensorrt\_tokenize的config.pbtxt配置文件



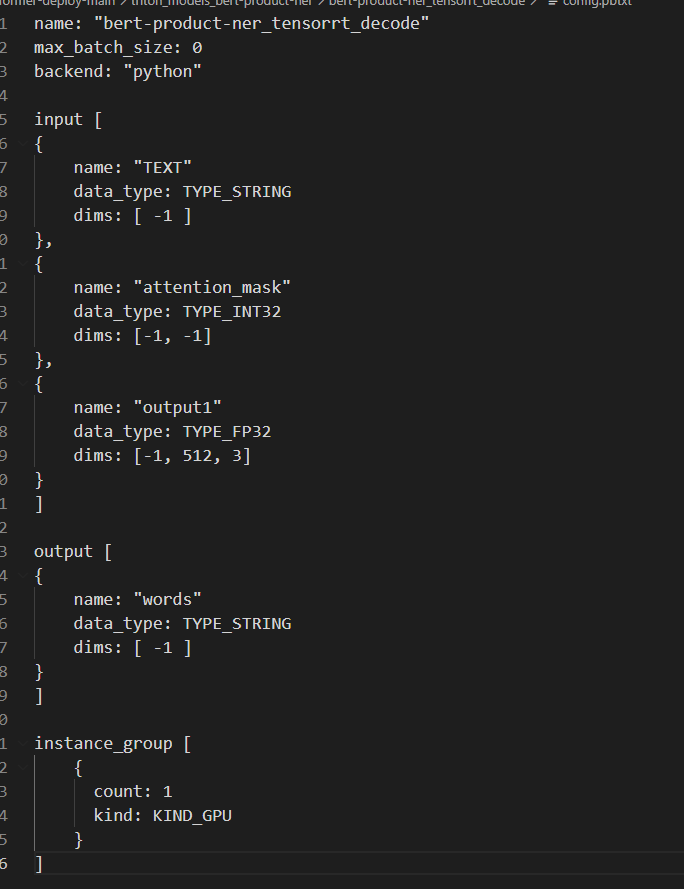
创建model.py文件，设置文本分词的代码逻辑，类名必须为TritonPythonModel，tokenizer的padding策略要和NER模型分词策略一致，返回的数据类型要和config.pbtxt配置文件定义的一致



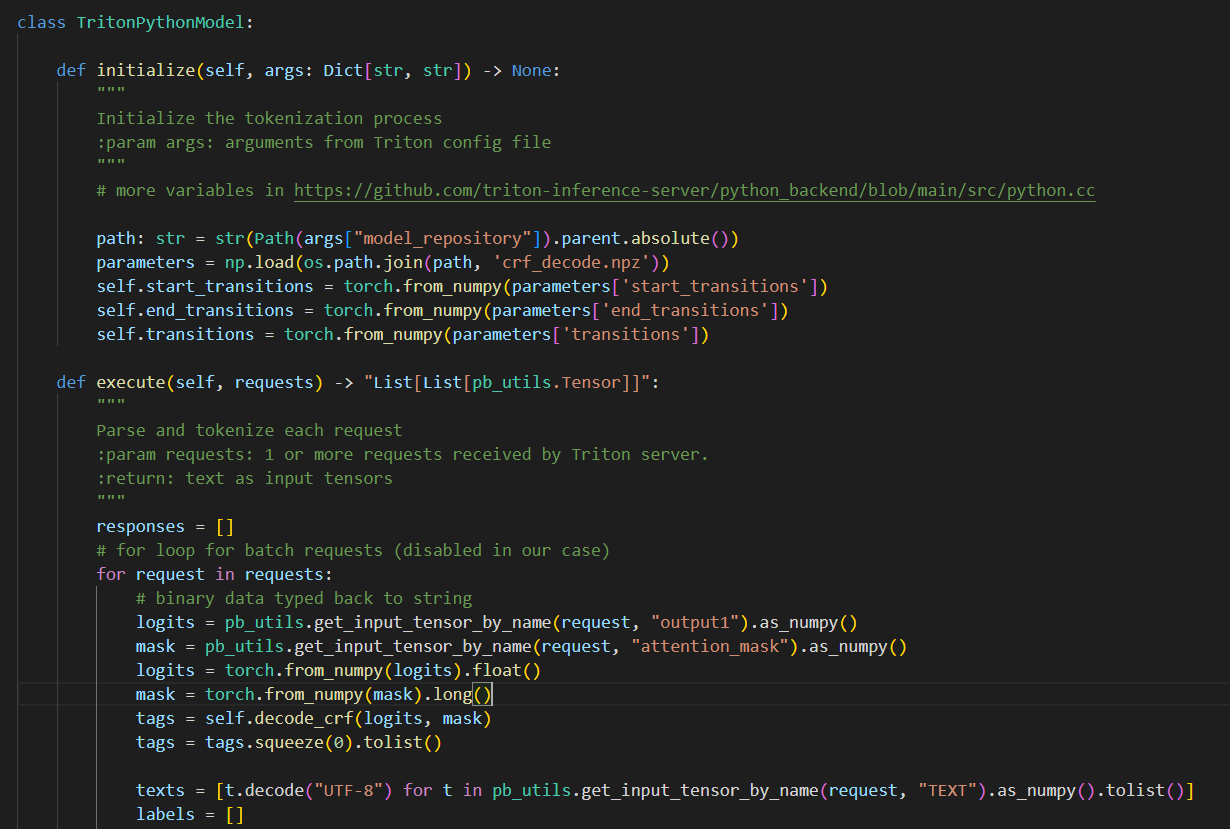
2.2 bert-ner推理任务的创建。利用tensorrt容器的trtexec将模型转成tensorrt格式，放入bert-product-ner\_tensorrt\_model文件夹。编写config.pbtxt模型配置文件定义输入和输出的字段。



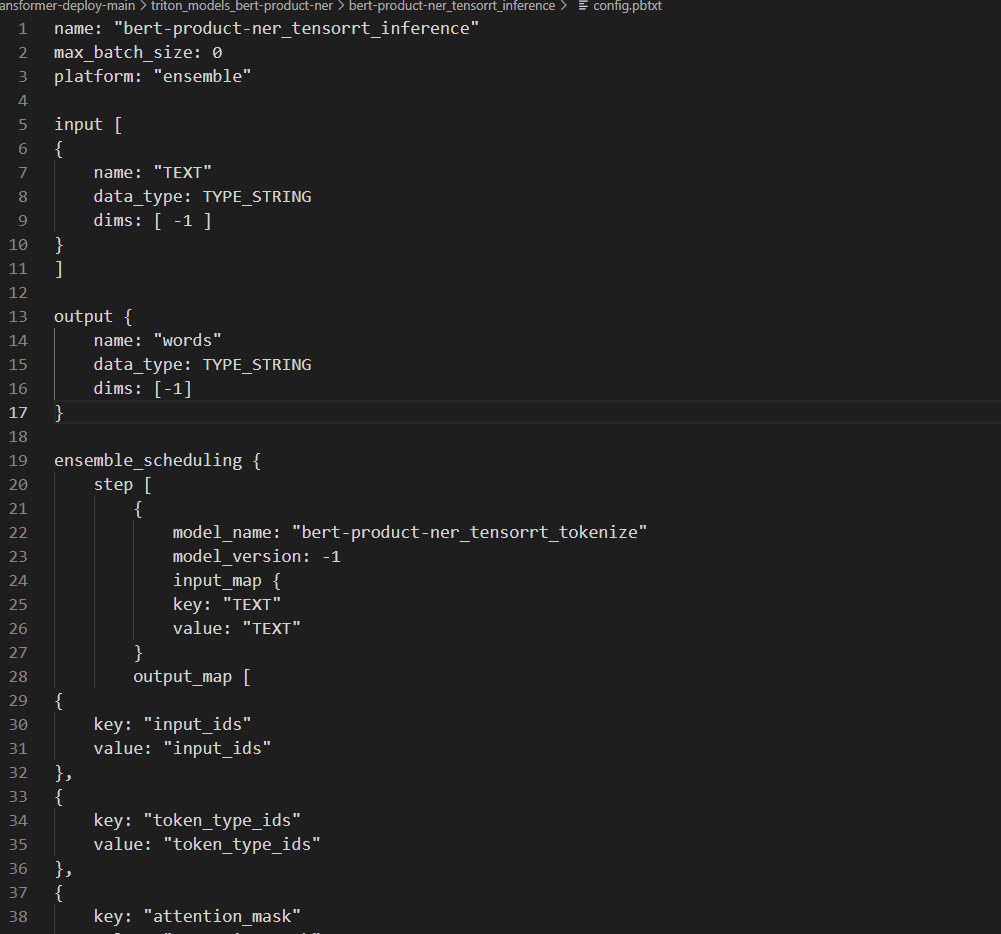
2.3 NER CRF解码任务的创建。在bert-product-ner\_tensorrt\_decode文件夹下创建config.pbtxt配置文件。



输入字段包含分词阶段的输入输出和bert-ner推理的输出，输出为识别的产品词数组。创建model.py文件编写crf decode逻辑。



2.4 Model pipeline任务创建，串连分词、bert-ner推理、CRF解码任务。在bert-product-ner\_tensorrt\_inference文件夹中创建config.pbtxt配置文件。其中step数组为每一阶段执行的任务。



2.5 启动服务，所有服务状态为READY并且出现8000、8001、8002端口表示服务启动成功。

model\_name='bert-product-ner'

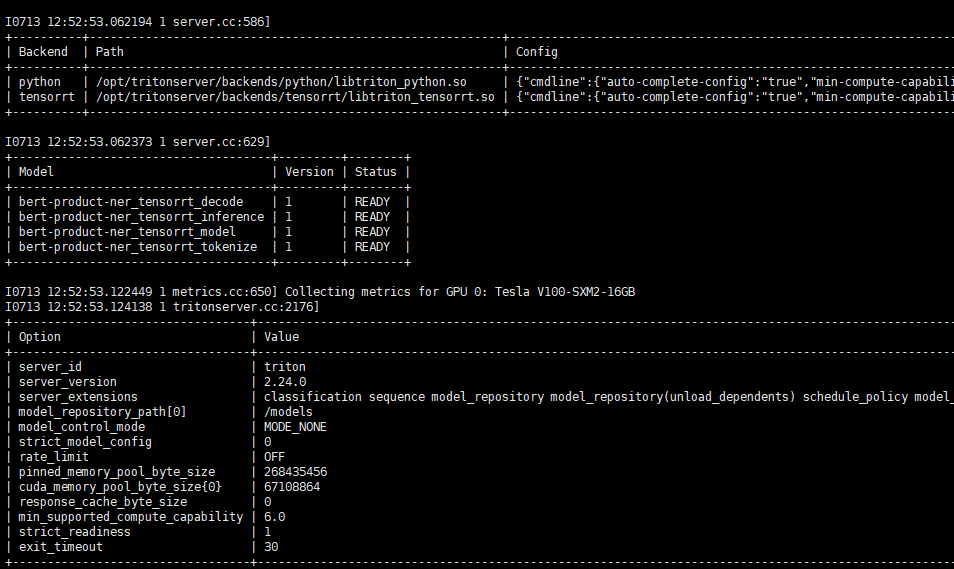
sudo docker run -it --rm --gpus all \

-p8003:8000 -p8004:8001 -p8005:8002 --shm-size 256m \

--name tritonserver \

-v $PWD/triton\_models\_${model\_name}:/models zuoxiaolei/tritonserver:22.07-py3 \

bash -c "tritonserver --model-repository=/models"



## **模型服务调用和测试**

**基于gRPC协议调用**

|  |
| --- |
| import tritonclient.http  import numpy as np  import time  import tritonclient.grpc  def test\_triton\_rpc(model='bert-product-classify',                       model\_type= "tensorrt",                       output\_names=['words']):      model\_name = f"{model}\_{model\_type}\_inference"      url = "127.0.0.1:8004"      model\_version = "1"      triton\_client = tritonclient.grpc.InferenceServerClient(url=url, verbose=False)      batch\_size = len(data)      query = tritonclient.grpc.InferInput(name="TEXT", shape=(batch\_size,), datatype="BYTES")      outputs = [tritonclient.grpc.InferRequestedOutput(name=ele) for ele in output\_names]      while 1:          query.set\_data\_from\_numpy(np.asarray(data, dtype=object))          start\_time = time.time()          query.set\_data\_from\_numpy(np.asarray(data, dtype=object))          response = triton\_client.infer(                  model\_name=model\_name, model\_version=model\_version, inputs=[query], outputs=outputs)          a= response.as\_numpy('words')          a =[ele.decode('utf-8') for ele in a]          print(time.time()-start\_time)          print(a)          print(len(a)) |

基于HTTP协议调用模型服务

|  |
| --- |
| def test\_triton\_client(data):      model\_type = "tensorrt"      model = 'bert-product-classify'      model\_name = f"{model}\_{model\_type}\_inference"      url = "127.0.0.1:8003"      model\_version = "1"      triton\_client = tritonclient.http.InferenceServerClient(url=url, verbose=False)      batch\_size = len(data)      query = tritonclient.http.InferInput(name="TEXT", shape=(batch\_size,), datatype="BYTES")      model\_score = tritonclient.http.InferRequestedOutput(name="output", binary\_data=False)      while 1:          query.set\_data\_from\_numpy(np.asarray(data, dtype=object))          start\_time = time.time()          query.set\_data\_from\_numpy(np.asarray(data, dtype=object))          response = triton\_client.infer(model\_name=model\_name, model\_version=model\_version, inputs=[query], outputs=[model\_score])          labels = response.as\_numpy('output')          print(time.time()-start\_time)          print(labels) |