OZON tech

Подробнее про triton backends

Виктор Горшков

Математик-разработчик

01

Что такое backends?
Для чего нужны?
Какие бывают?

Что такое triton backend

Это модули в NVIDIA Triton Inference Server, которые определяют, как именно сервер будет обрабатывать запросы для конкретной модели.

Каждый бэкенд отвечает за работу определённого типа моделей и предоставляет API для обработки входных данных, выполнения вычислений и получения результатов.

Позволяет гибко обрабатывать запросы, масштабировать выполнение моделей и ускорять работу, используя различные технологии и подходы.

02

Какие бэкенды рассмотрим поверхностно?

Множество бэкендов

Backend	Назначение	Актуальность
Dali	Оптимизация процесса предобработки данных	Да
Python	Описание разнородной логики и связующая часть моделей в ансамбле	Да
PyTorch	Запуск PyTorch моделей без особых оптимизаций	Да
TensorFlow	Запуск TensorFlow моделей без особых оптимизаций	Да
FIL	Запуск моделей на основе деревьев	Да
vLLM	Быстрое резвёртыване языковых моделей	Да
ONNX Runtime	Запуск моделей в ONNX	Да
TensorRT-LLM	Оптимизация вычислений языковых моделей под NVIDIA GPU	Да
TensorRT	Оптимизация вычислений непосредственно под NVIDIA GPU	Да
OpenVino	Оптимизация моделей для запуска на CPU	Да
C++	Реализация логики на C++	Нет (Неактуально/сложно)
Custom	Инструмент для создания нового бэкенда	Нет (Неактуально/сложно)
Stateful	Бэкенд с сохранением промежуточных состояний	Нет (Не поддерживается)
Square	Выполнение матричных преобразований	Нет (Неактуально)
Faster Transformer	Запуск моделей на основе трансформеров	Нет (Устарело)

03

Какие бэкенды рассмотрим детально?

Dali Backend

Предназначен для оптимизации и ускорения предварительной обработки данных.

Используется для обработки изображений, видео и других данных непосредственно на GPU, но также оставляет режим работы на CPU.

Преимущество

При запуске на GPU выполняет преобразования, используя NPP (NVIDIA Performance Primitives) за счёт чего получается выигрыш в скорости.

Недостаток

В NPP реализация некоторых преобразований отличается их аналогов, например, в OpenCV и PIL. Это отличие влечёт за собой отличие в результате. Этот факт **нужно** учитывать при обучении моделей.

Определение pipeline dali

*.py

```python
import os

import numpy as np
import nvidia.dali.fn as fn
import nvidia.dali.types as types
from nvidia.dali import pipeline_def

SIZE_X, SIZE_Y, SCALE, OUTPUT_DIR = …, …, …, …
STD, MEAN = […, …, …], […, …, …]

@pipeline_def(batch_size=16, num_threads=4)
def TritonLoadingPipeline():
    img_files = fn.external_source(name="DALI_INPUT_0")

    orig_images = fn.decoders.image(
        img_files,
        device="cpu"
    )

    resized_images = fn.resize(
        orig_images,
        resize_x=SIZE_X,
        resize_y=SIZE_Y,
        device="gpu",
    )

    normalized_images = fn.crop_mirror_normalize(
        resized_images,
        mean=MEAN,
        std=STD,
        scale=SCALE,
        output_layout="CHW",
        dtype=types.FLOAT,
        device="gpu",
    )
    return normalized_images


if __name__ == "__main__":
    pipe = TritonLoadingPipeline(device_id=0)
    output_fn = os.path.join(OUTPUT_DIR, "model.dali")
    pipe.serialize(filename=output_fn)
```

При выборе device работает нормально последовательность cpu -> gpu.

Если на каком-то шаге будет переход gpu -> cpu, pipeline либо не пройдет сериализацию, либо этап сборки в тритоне.

Про различные методы Dali pipeline можно дополнительно почитать тут.

Сборка модели

DALI Backend должен быть включен в Triton Inference Server. Для этого используется контейнер Triton с поддержкой DALI.

Например, nvcr.io/nvidia/tritonserver:23.10-py3

Название файла

```
models_repository
└── dali_model
    ├── 1
    │   └── model.dali
    └── config.pbtxt
```

config.pbtxt

```
name: "dali_model"
backend: "dali"
max_batch_size: 64
input [
  {
    name: "DALI_INPUT_0"
    data_type: TYPE_STRING
    dims: [ -1 ]
  }
]
output [
  {
    name: "DALI_OUTPUT_0"
    data_type: TYPE_FP32
    dims: [ 3, 224, 224 ]
  }
]
```

Python Backend

Предназначен для реализации произвольной логики. Позволяет быстро вносить изменения и выступает как соединительное звено между частями ансамбля.

Структура

```
model_repository
└── python_model
    ├── 1
    │   ├── model.py
    │   ├── utils.py
    │   └── …
    └── config.pbtxt
```

Рядом с `model.py`, могут лежать другие необходимые файлы и каталоги с кодом или артефактами в виде pth, yaml, и т.д

config.pbtxt

```
name: "python_model"
backend: "python"
max_batch_size: 1
input [
  {
    name: "INPUT_0"
    data_type: TYPE_FP32
    dims: [ -1 ]
  },
  {
    name: "INPUT_1"
    data_type: TYPE_FP32
    dims: [ -1 ]
  }
]
output [
  {
    name: "OUTPUT_0"
    data_type: TYPE_FP32
    dims: [ -1 ]
  },
  {
    name: "OUTPUT_1"
    data_type: TYPE_FP32
    dims: [ -1 ]
  }
]
parameters: [{
    key: "EXECUTION_ENV_PATH",
    value: {
        string_value: "$$TRITON_MODEL_DIRECTORY/conda_env.tar.gz"
    }
}]
```

Пример модели

model.py

```python
import json
import triton_python_backend_utils as pb_utils

class TritonPythonModel:
    def initialize(self, args):
        self.model_config = json.loads(args["model_config"])

        output0_config = pb_utils.get_output_config_by_name(model_config, «OUTPUT_0")
        output1_config = pb_utils.get_output_config_by_name(model_config, «OUTPUT_1")

        self.output0_dtype = pb_utils.triton_string_to_numpy(
            output0_config["data_type"]
        )
        self.output1_dtype = pb_utils.triton_string_to_numpy(
            output1_config["data_type"]
        )

    def execute(self, requests):
        responses = []

        for request in requests:
            in_0 = pb_utils.get_input_tensor_by_name(request, «INPUT_0")
            in_1 = pb_utils.get_input_tensor_by_name(request, «INPUT_1")

            out_0, out_1 = (
                in_0.as_numpy() + in_1.as_numpy(),
                in_0.as_numpy() - in_1.as_numpy(),
            )

            out_tensor_0 = pb_utils.Tensor(«OUTPUT_0", out_0.astype(self.output0_dtype))
            out_tensor_1 = pb_utils.Tensor(«OUTPUT_1", out_1.astype(self.output1_dtype))

            inference_response = pb_utils.InferenceResponse(
                output_tensors=[out_tensor_0, out_tensor_1]
            )
            responses.append(inference_response)

        return responses

    def finalize(self):
        print("Модель успешно выгрузилась!")
```

Initialize

В `initialize` допустимо загружать веса модели, создавать её экземпляр и инициализировать другие переменные, чтобы сократить время выполнения метода execute.

execute

В методе `execute` моделируется некоторая логика, которая будет выполняться, когда модель придут одиночные запросы или батчи. Важно иметь в виду, что этот метод каждый раз выполняется, когда приходит запрос.

Вычислительная сложность именно этого метода будет влиять на скорость отклика модели.

finalize

В эпоху существования сборщиков мусора метод `finalize` немного потерял свой основной вес, но тем не менее по-прежнему существует ряд задач, которые может потребоваться решать перед выгрузкой, например, логирование момента выгрузки или состояний.

Примеры ситуаций

Ситуация 1

В репозитории production сервера уже загружены и работают две модели. Необходимо развернуть третью модель, которой нужны дополнительные пакеты. Пересборка образа не допустима, т.к. она влечёт существенные затраты по времени.

Решение

Нужно создать переносимое окружение для третьей модели и совместно загрузить их в репозиторий сервера. Triton-сервер для запуска модели будет использовать отдельный интерпретатор со своим набором пакетов, никак не влияя на уже работающие модели.

Структура

```
models
└── python_model
    ├── 1
    │   └── model.py
    ├── config.pbtxt
    ├── conda_env_3_9.tar.gz
    └── triton_python_backend_stub
```

Ситуация 2

Для новой модели устанавливается набор библиотек в среде python 3.9. Серверная python-оболочка скомпилирована с использованием Python 3.10. Получается, что этот набор библиотек не будет доступен в новой модели.

Решение

Для решения проблемы, кроме собора портативного окружения, компилируется **triton_python_backend_stub** и подкладывается вместе с окружением в репозиторий модели.

Сборка окружения

bash

```
export CONDA_SOURCE=https://repo.anaconda.com/miniconda/
Miniconda3-latest-Linux-x86_64.sh
export CONDA_INSTALLER=Miniconda3.sh
export SAVE_FOLDER=/tmp/conda_env
export ENV_NAME=conda_env_3_9
export PYTHON_VERSION=3.9

wget $CONDA_SOURCE -O $SAVE_FOLDER/$CONDA_INSTALLER
chmod +x $SAVE_FOLDER/$CONDA_INSTALLER
bash $CONDA_INSTALLER -b -p $SAVE_FOLDER/miniconda
conda init bash
conda create -ny $ENV_NAME python=$PYTHON_VERSION
conda activate $ENV_NAME
conda install -y conda-pack pip
...
conda pack -fo $SAVE_FOLDER/$ENV_NAME.tar.gz
...
conda deactivate
rm -r $SAVE_FOLDER
```

В контейнере Triton для работы с моделями, реализованными на Python, присутствует python-интерпретатор. Версия этого интерпретатора зависит от версии серверной части Triton и сборки самого контейнера.

Безусловно, можно использовать базовый интерпретатор, но ровно до тех пор, пока не появится потребность в разных наборах библиотек для разных моделей на python-бэкенде.

Часто необходимо использовать библиотеку или фреймворк, которых нет на уже запущенном сервере.

Перезапускать production сервер для установки дополнительных пакетов — не оправдано, как минимум, из-за высокого риска конфликта версий библиотек.

Библиотеки часто могут не поддерживать работу с текущей версией интерпретатора.

Для исключения подобных проблем Triton-сервер имеет поддержку виртуальных окружений (virtualenv, conda environment).

В случаях, когда версия интерпретатора в переносимом окружении отличается от версии интерпретатора triton-сервера, для исправной работы модели потребуется скомпилировать новый файл-заглушку **triton_python_backend_stub**.

Этот файл является связующим звеном между интерпретатором переносимого окружения и скомпилированным ядром Triton, написанном на C++.

О сборке triton_python_backend_stub можно прочитать в официальной документации.

FIL (Forest Inference Library) Backend

Предназначен для выполнения моделей на основе деревьев: XGBost, LightGBM и случайные леса из Scikit-Learn и cuML.

Структура

```
model_repository
└── fil_model
    ├── 1
    │   └── model.(xgboost, txt, treelite, treelite.json)
    └── config.pbtxt
```

model_type — тип модели: xgboost_json, lightgbm, treelite_json.

output_class — включает возвращение метку класса.

threshold — порог для классификации.

config.pbtxt

```
name: "fil_model"
backend: "fil"
max_batch_size: 64
input [
  {
    name: "input__0"
    data_type: TYPE_FP32
    dims: [ -1 ]
  }
]
output [
  {
    name: "output__0"
    data_type: TYPE_FP32
    dims: [ -1 ]
  }
]

parameters: {
  key: "model_type"
  value: { string_value: "xgboost_json" }
}
parameters: {
  key: "output_class"
  value: { string_value: "true" }
}
parameters: {
  key: "threshold"
  value: { string_value: "0.5" }
}
```

Пример создания модели

```python
import xgboost as xgb

dtrain = xgb.DMatrix(
    data=[[1, 2], [3, 4]],
    label=[0, 1]
)
params = {
    "objective": "binary:logistic",
    "max_depth": 2,
    "eta": 1,
    "tree_method": "gpu_hist",
}
bst = xgb.train(
    params, dtrain,
    num_boost_round=10
)
bst.save_model("model.xgboost")
```

Структура

```
model_repository
└── fil_model
    ├── 1
    │   └── model.(xgboost, txt, treelite, treelite.json)
    └── config.pbtxt
```

Таким образом, в зависимости от фреймворка, в модели будут разные расширения сериализованных моделей.

XGBoost: model.xgboost (бинарный формат)

LightGBM: model.txt (текстовый формат)

Treelite: model.treelite или model.treelite.json

PyTorch Backend

Предназначен для выполнения моделей PyTorch, сохранённых в формате **TorchScript**.

Структура

```
models_repository
└── pytorch_model
    ├── 1
    │   └── model.pt
    └── config.pbtxt
```

Пример создания модели

```python
import torch
import torch.nn as nn


class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc = nn.Linear(5, 1)

    def forward(self, x):
        return torch.sigmoid(self.fc(x))


model = SimpleModel()
scripted_model = torch.jit.script(model)
scripted_model.save("model.pt")
```

config.pbtxt

```
name: "pytorch_model"
platform: "pytorch_libtorch"
max_batch_size: 8
input [
  {
    name: "INPUT__0"
    data_type: TYPE_FP32
    dims: [ 5 ]
  }
]
output [
  {
    name: "OUTPUT__0"
    data_type: TYPE_FP32
    dims: [ 1 ]
  }
]
instance_group [
  {
    kind: KIND_GPU
  }
]
```

TensorFlow Backend

Предназначен для запуска моделей, сохранённых в формате **SavedModel** или **Frozen Graph** на CPU или GPU.

Структура

```
models_repository
└── tensorflow_model
    ├── 1
    │   ├── variables/
    │   └── model.pb
    └── config.pbtxt
```

TensorFlow SavedModel: директория содержит файлы saved_model.pb и подкаталог variables/.

В config.pbtxt указывается *tensorflow_savedmodel.*

TensorFlow Frozen Graph: директория содержит один файл .pb.

В config.pbtxt указывается *tensorflow_graphdef.*

Пример создания модели

```python
import tensorflow as tf


model = tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape=(28, 28, 1)),
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu'),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.save("model_repository/tensorflow_model/1/")
```

config.pbtxt

```
name: "tensorflow_model"
platform: "tensorflow_savedmodel"
max_batch_size: 8
input [
  {
    name: "INPUT__0"
    data_type: TYPE_FP32
    dims: [ 5 ]
  }
]
output [
  {
    name: "OUTPUT__0"
    data_type: TYPE_FP32
    dims: [ 1 ]
  }
]
instance_group [
  {
    kind: KIND_GPU
  }
]
```

vLLM Backend

Предназначен для упрощения запуска больших языковых моделей (LLM), таких как GPT, OPT, LLaMA, с использованием токен-кэширования и GPU.

Структура

```
models_repository
└── vllm_model
    ├── 1
    │   ├── config.json
    │   ├── merges.txt
    │   ├── special_tokens_map.json
    │   ├── tokenizer.json
    │   ├── generation_config.json
    │   ├── tokenizer.json
    │   ├── model.safetensors
    │   ├── vocab.json
    │   └── tokenizer_config.json
    └── config.pbtxt
```

Впрочем набор фалов может быть иным.

В каталог убирается всё, что появилось после скачивания или сохранения модели.

vLLM Backend

Подготовка модели

```python
from transformers import AutoTokenizer, AutoModelForCausalLM


model_name = "facebook/opt-125m"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

save_path = "./vllm_model"
tokenizer.save_pretrained(save_path)
model.save_pretrained(save_path)
```

config.pbtxt

```
name: "vllm_model"
backend: "vllm"
max_batch_size: 1
input [
  {
    name: "input_text"
    data_type: TYPE_STRING
    dims: [ -1 ]
  }
]
output [
  {
    name: "output_text"
    data_type: TYPE_STRING
    dims: [ -1 ]
  }
]
parameters: {
  key: "model_checkpoint"
  value: { string_value: "/models/vllm_model/1" }
}
parameters: {
  key: "max_input_length"
  value: { string_value: "512" }
}
parameters: {
  key: "max_output_length"
  value: { string_value: "128" }
}
```

ONNX Runtime Backend

Предназначен для выполнения моделей, экспортированных в формате ONNX (Open Neural Network Exchange). Это универсальный бэкенд, который поддерживает модели, созданные в различных фреймворках, таких как PyTorch, TensorFlow и другие.

Структура

```
models_repository
└── dali_model
    ├── 1
    │   └── model.onnx
    └── config.pbtxt
```

Официальный репозиторий - onnxruntime_backend

Конвертация в ONNX Runtime

```python
import torch

class SimpleModel(torch.nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc = torch.nn.Linear(5, 1)

    def forward(self, x):
        return torch.sigmoid(self.fc(x))

model = SimpleModel()
dummy_input = torch.randn(1, 5)
torch.onnx.export(model, dummy_input, "model.onnx")
```

config.pbtxt

```
name: "onnx_model"
platform: "onnxruntime_onnx"
max_batch_size: 16
input [
  {
    name: "input"
    data_type: TYPE_FP32
    dims: [ 5 ]
  }
]
output [
  {
    name: "output"
    data_type: TYPE_FP32
    dims: [ 1 ]
  }
]
instance_group [
  {
    kind: KIND_GPU
  }
]
```


Официальный репозиторий - onnxruntime_backend

OpenVINO Backend

Предназначен для оптимизации и ускорения .

Структура

```
models_repository
└── openvino_model
    ├── 1
    │   ├── model.bin
    │   └── model.xml
    └── config.pbtxt
```

OpenVINO Backend поддерживает работу только она CPU от Intel и arm.

На данный момент в OpenVINO конвертацию поддерживают:

- PyTorch
- TensorFlow
- TensorFlow Lite
- ONNX
- PaddlePaddle
- JAX/Flax
- OpenVINO IR

Итоговая модель после конвертации состоит из двух файлов: .xml, содержащего информацию о топологии сети, и файл .bin, содержащего данные о весах.

Конвертация в OpenVINO

Из onnx

```python
import openvino as ov

ONNX_NLP_MODEL_PATH = ...
MODEL_DIRECTORY_PATH = ...

ov_model = ov.convert_model(
    ONNX_NLP_MODEL_PATH,
    input=[("input", [1, 3, 224, 224])]
)
ov.save_model(
    ov_model, MODEL_DIRECTORY_PATH / "model.xml",
    compress_to_fp16=False
)
```

config.pbtxt

```
name: "openvino_model"
platform: "openvino_ir"
max_batch_size: 8
input [
  {
    name: "input"
    data_type: TYPE_FP32
    dims: [ 1, 3, 224, 224 ]
  }
]
output [
  {
    name: "output"
    data_type: TYPE_FP32
    dims: [ 1, 1000 ]
  }
]
```

Подробнее с OpenVINO можно познакомиться тут.

А непосредственно про OpenVINO Backend можно почитать тут.

TensorRT-LLM Backend

Предназначен для оптимизации и ускорения работы языковых моделей.

Содержит внутри кэширование токенов, ускорения для последовательной обработки токенов и иные оптимизации, а также упрощения для настройки.

Структура

```
models_repository
└── tensorrt_llm_model
    ├── 1
    │   └── model.plan
    └── config.pbtxt
```

TensorRT-LLM Backend

Конвертация

```
transformers-cli download facebook/opt-125m

trtllm-build \
  --hf-model ./facebook_opt-125m \
  --engine ./model.plan \
  --max-seq-len 512 \
  --precision fp16 \
  --batch-size 8
```

config.pbtxt

```
name: "trt_llm_model"
backend: "tensorrt_llm"
max_batch_size: 8

input [
  {
    name: "input_ids"
    data_type: TYPE_INT32
    dims: [ -1 ]
  },
  {
    name: "attention_mask"
    data_type: TYPE_INT32
    dims: [ -1 ]
  }
]
output [
  {
    name: "output_ids"
    data_type: TYPE_INT32
    dims: [ -1 ]
  }
]
parameters: {
  key: "model_checkpoint"
  value: { string_value: "/models/trt_llm_model/1/
model.plan" }
}
parameters: {
  key: "max_input_length"
  value: { string_value: "512" }
}
parameters: {
  key: "max_output_length"
  value: { string_value: "128" }
}
```


Официальный репозиторий - tensorrtllm_backend

TensorRT Backend

Предназначен для запуска моделей, оптимизированных с помощью NVIDIA TensorRT, в формате TensorRT Engine (.plan). Этот бэкенд подходит для любых задач, где требуется максимальная производительность на GPU.

Структура

```
models_repository
└── tensorrt_model
    ├── 1
    │   └── model.plan
    └── config.pbtxt
```

TensorRT Backend

Конвертация

```
import tensorrt as trt

TRT_LOGGER = trt.Logger(trt.Logger.WARNING)
with trt.Builder(TRT_LOGGER) as builder:
    network = builder.create_network()
    parser = trt.OnnxParser(network, TRT_LOGGER)

    with open("model.onnx", "rb") as model_file:
        parser.parse(model_file.read())

    engine = builder.build_cuda_engine(network)

# Или с помощью cli утилиты
!trtexec --onnx=model.onnx --saveEngine=model.plan --fp16
```

config.pbtxt

```
name: "tensorrt_model"
platform: "tensorrt_plan"
max_batch_size: 32
input [
  {
    name: "input"
    data_type: TYPE_FP16
    dims: [ 3, 224, 224 ]
  }
]
output [
  {
    name: "output"
    data_type: TYPE_FP16
    dims: [ 1000 ]
  }
]
```

TensorRT vs TensorRT-LLM

Характеристика	TensorRT	TensorRT-LLM
Назначение	Универсальный инференс моделей	Инференс больших языковых моделей (LLM)
Типы задач	GPU	GPU С дополнительными оптимизациями для LLM
Оптимизация для LLM	Нет	Да
Типы моделей	Любые типы	LLM (GPT, OPT, LLaMA, T5)
Тип входных данных	Изображения, видео, массивы	Токены и маски внимания
Пример приложений	YOLO, ResNet, U-Net	GPT-3, BERT, чат-боты
Формат моделей	TensorRT Plan	TensorRT Plan
Оборудование	GPU	GPU с бОльшим объёмом памяти
Преимущества	Широкий спектр задач, высокая производительность	Высокая производительность для LLM, поддержка токен-кэширования
Недостатки	Оптимизация вычислений под NVIDIA GPU	Оптимизация вычислений под NVIDIA GPU
Утилита	trtexec	trtllm-build

Немного про конфиги

backend	platform
dali	Нет
python	python
fil	xgboost, lightgbm, treelite
pytorch	pytorch_libtorch
tensorflow	tensorflow_savedmodel, tensorflow_graphdef
onnxruntime	onnxruntime_onnx
openvino	openvino_ir
Нет	tensorrt_llm
tensorrt	tensorrt_plan
ensemble	Нет

04

Какой когда использовать?

Сравнительная таблица

Backend	Оборудование	Назначение
Dali	CPU/GPU	Оптимизация процесса предобработки данных.
Python	CPU/GPU	Различная логика, неумещающаяся в остальные форматы.
FIL	CPU/GPU	Оптимизация вычислений для древовидных моделей.
PyTorch	CPU/GPU	Упрощенный запуск PyTorch моделей.
TesorFlow	CPU/GPU	Упрощенный запуск TF моделей.
vLLM	CPU/GPU	Упрощенный запуск LLM на основе Python backend.
ONNX Runtime	CPU/GPU	Запуск моделей в ONNX.
TensorRT-LLM	GPU	Оптимизация вычислений для LLM под NVIDIA GPU.
OpenVINO	CPU	Оптимизация моделей для запуска без GPU.
TensorRT	GPU	Оптимизация вычислений под NVIDIA GPU.

05

Важные моменты

Точность конвертирования

Во всех случаях, когда будет преобразования типов данных, стоит иметь ввиду, что значения выдаваемые моделями будут отличаться.

Во всех случаях, когда есть аппаратно зависимые оптимизации, также стоит ожидать отличия в значениях.

Главный вопрос — насколько это отличие влияет на итоговый вердикт модели.

Поддержка бэкендов

Не все бэкенды поддерживаются каждой версией тритон сервера, кроме того они поддержка зависит от ОС.

Актуальное по версиям можно посмотреть в документации или в репозитории, переключаясь по веткам.

Назначение образов

В docker репозитории nvidia уже есть готовые окружения с тритон сервером, нужно выбрать подходящий под свои нужды.

Небольшое описание.

OZONtech

Всё!

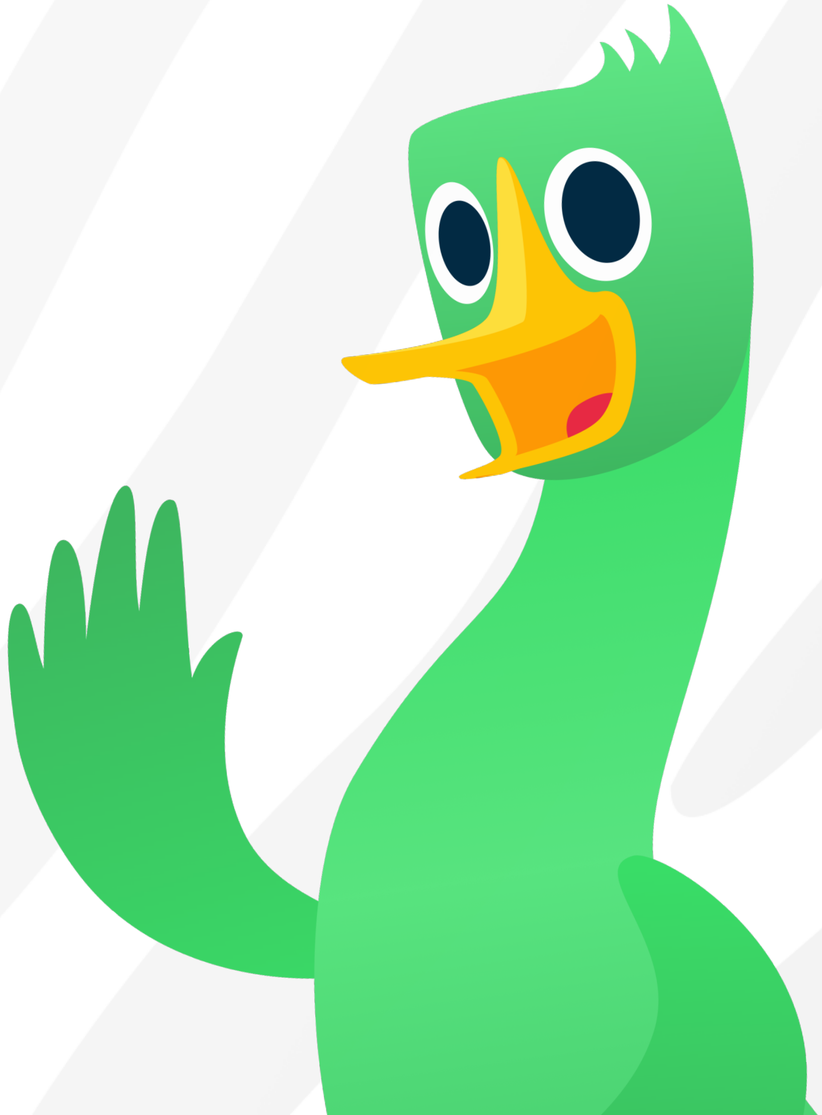
Виктор Горшков

Математик-разработчик

tg: @Victor_Gorshkov

06

Задание



Задание «Пробуем бэкенды и готовим модель к проду»

1. Нужно взять веса модели в pth тут.
2. Сконвертировать модель в onnx fp32.
3. Квантизировать до onnx fp16.
4. * *Сконвертировать модель в TensorRT из onnx fp16.*
5. Через 3(4) полученных варианта моделей пропустить набор тестовых данных и получить логиты положительного класса моделей * *(Можно прогонять данные через клиент в поднятый тритон, тоже за отдельные бонусы)*.
6. Зафиксировать порог такой, чтобы обеспечить precision 0.9 по результатам для модели pth.
7. Проверить, не отличаются ли вердикты полученные по выбранному порогу, между моделью в pth, в onnx fp32 и в onnx fp16 и (tensorRT).
8. Посчитать метрики:
   * Среднее расхождение между моделями по логитам, а также абсолютные максимальные и минимальные значения отклонения;
   * Считая вердикты модели в pth верными, построить матрицу ошибок для в onnx fp32 и onnx fp16, (TensorRT), посчитать accuracy, precision и recall.
9. Если отличаются, то для моделей с расхождением подобрать пороги, обеспечивающие наиболее близкие вердикты к pth.
10. Сделать вывод о том, какую модель стоит отправить в прод. Объяснить решение.
11. Собрать файл triton-модели.