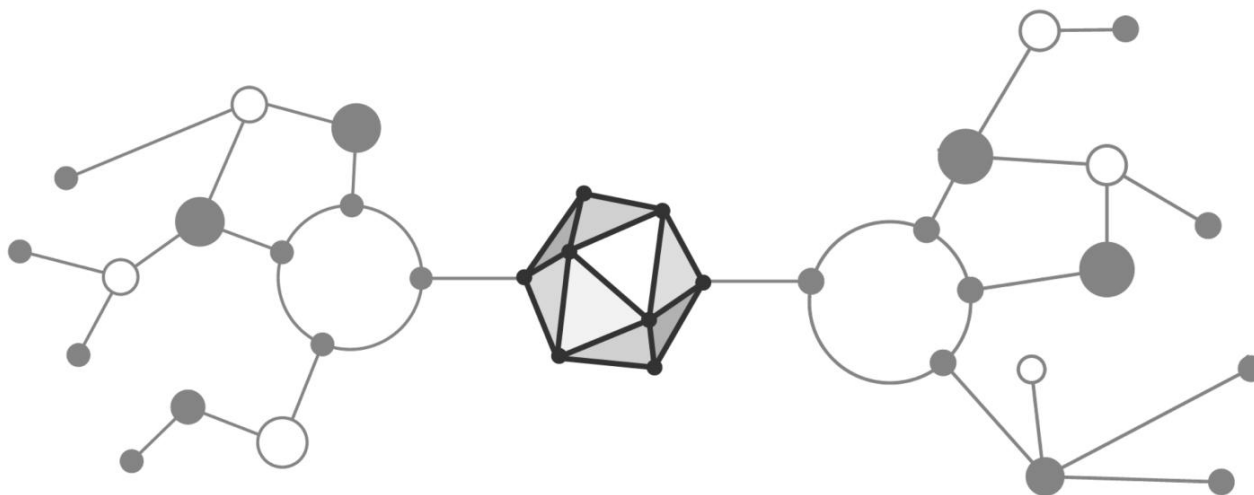


ONNX

Open Neural Network Exchange

ONNX



开放式神经网络交换 (ONNX) 是一个开放式生态系统。

ONNX 定义了一种**通用的文件格式**，用于存储已训练好的模型。

ONNX 提供了可扩展计算图模型的定义，以及内置运算符和标准数据类型的定义。

ONNX 文件不仅存储了**神经网络模型的权重**，同时也存储了**模型的结构信息**以及**网络中每一层的输入输出**和一些其它的辅助信息。

ONNX支持的深度学习框架



Transformers



Keras

LibSVM

MATLAB®

[M]^s MindSpore



PaddlePaddle



SIEMENS



SciKit Learn



支持的推理架构

Optimize Inferencing		Optimize Training														
Platform	Windows		Linux		Mac		Android		iOS		Web Browser (Preview)					
API	Python		C++		C#		C		Java		JS		Obj-C		WinRT	
Architecture	X64			X86			ARM64			ARM32			IBM Power			
Hardware Acceleration	Default CPU			CoreML			CUDA			DirectML			oneDNN			
	OpenVINO			TensorRT			NNAPI			ACL (Preview)			ArmNN (Preview)			
	MIGraphX (Preview)			Rockchip NPU (Preview)			SNPE			TVM (Preview)			Vitis AI (Preview)			
Installation Instructions		Please select a combination of resources														

ONNX 安装:

CPU版:

pip install onnxruntime

GPU版本:

pip install onnxruntime-gpu

导出模型安装 ONNX:

深度学习框架	为模型导出安装 ONNX
pytorch	torch
tensorflow	tf2onnx
paddlepaddle	paddle2onnx
sklearn	skl2onnx
keras	keras2onnx onnxconverter-common
CLIP	clip-onnx

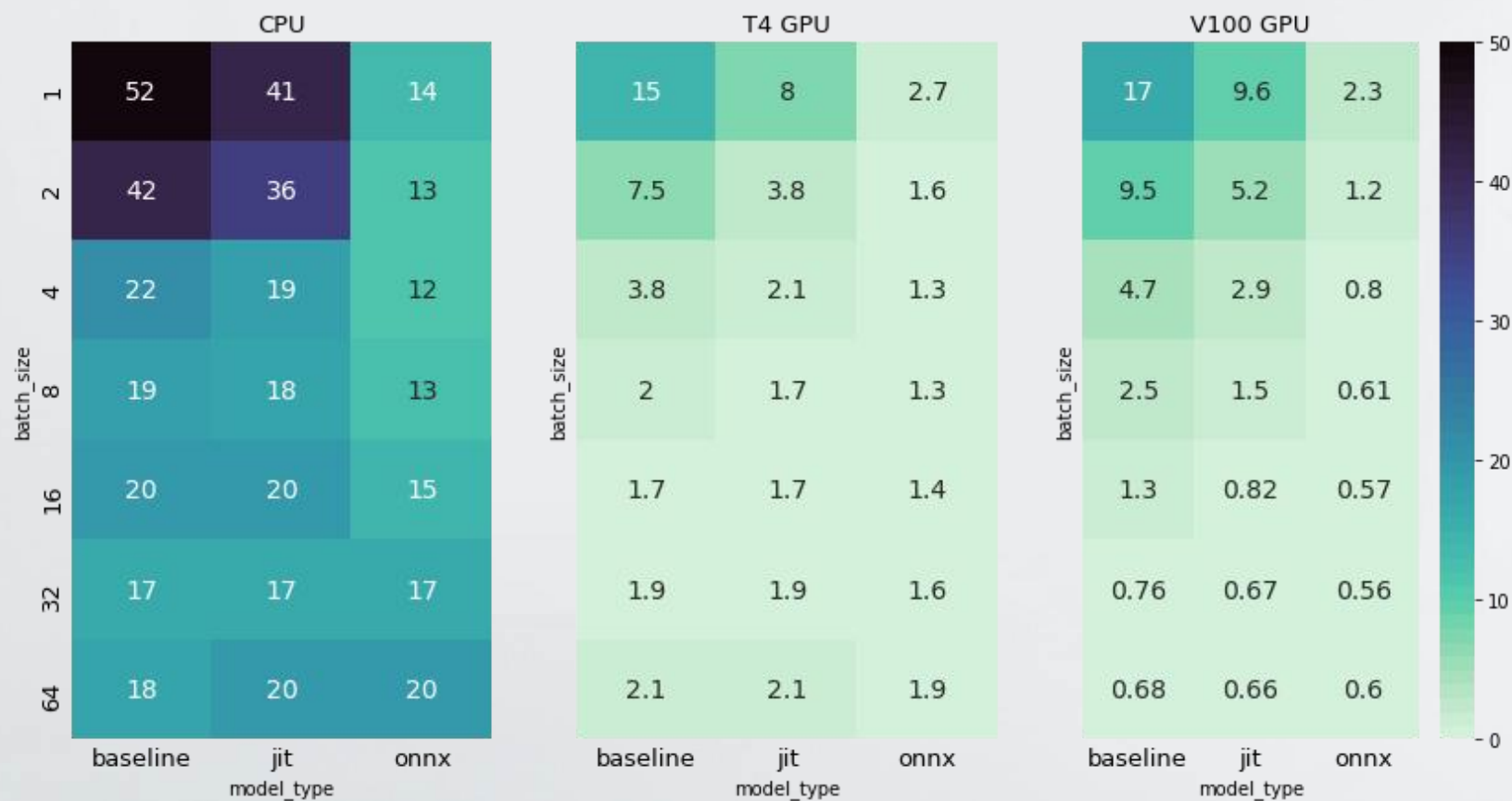


ONNX 和 Pytorch 推理加速对比

- 基于bert变体的分类实验

1. 不同机器的推理速度对比

Mean inference time in ms per sequence



2. 不同序列长度对比

Mean inference time in ms per sequence on V100 GPU



结论:

1. 对于CPU而言, batch_size小于32时, onnx格式更强大;
2. 对于不同序列长度而言, 单个样本加速效果onnx更明显;

ONNX 模型转换

1. NLP 以 Bert 为例:

```
import torch, transformers
from transformers import BertForSequenceClassification
```

- Pytorch 自带的 `torch.onnx.export` 方法:

1. 加载预训练模型: (分类任务)

```
model = BertForSequenceClassification.from_pretrained(pytorch_model)
```

2. 不启用 **Batch Normalization** 和 **Dropout**:

```
model.eval()
```

3. 设置示例向量:

```
inputs = { 'input_ids': torch.ones((1, 512), dtype=torch.long),
           'attention_mask': torch.ones((1, 512), dtype=torch.long),
           'token_type_ids': torch.zeros((1, 512), dtype=torch.long)}
```

4. 不进行反向传播:

```
with torch.no_grad():
```



5. 动态轴设置:

```
symbolic_names = {0: 'batch_size', 1: 'max_seq_len'}
```

6. 转换模型:

```
torch.onnx.export(  
    model, torch模型文件夹路径  
    (inputs['input_ids'], 模型输入（或多个输入的元组）  
    inputs['attention_mask'],  
    inputs['token_type_ids']),  
    onnx_model, 导出模型文件路径  
    opset_version=11, ONNX算子版本  
    do_constant_folding=True, 常量折叠优化  
    input_names=['input_ids', 模型输入张量名称  
                'input_mask',  
                'segment_ids'],  
    output_names=['bios'], 模型张量输出名称  
    dynamic_axes={'input_ids': symbolic_names,  
                  'input_mask': symbolic_names,  
                  'segment_ids': symbolic_names,  
                  'bios': [0]} 动态轴设置并添加输出  
    )
```



● 使用 huggingface 的 transformers 提供的模型转换工具

1. 转换模型:

```
transformers.convert_graph_to_onnx.convert(  
    framework="pt", 选择torch或者tensorflow模型类型  
    model=origin_model_path, 待转换的模型文件（夹）路径  
    output=Path("onnx/onnx_model.onnx"), 导出模型文件位置  
    opset=11 导出的ONNX版本（算子对应）  
)
```

转换结果:

 onnx_model.onnx	2022/8/22 19:49	ONNX 文件	399,588 KB
---	-----------------	---------	------------

分析:

1. transformers 提供的 API 封装程度较高，底层还是调用的 torch.onnx.export 方法;
2. 对于自定义任务，无法使用 transformers.convert_graph_to_onnx.convert，需要用 torch.onnx.export 导出 ONNX 模型;

注意：

1. 为什么要设置动态轴？

这个会影响到后续转换 TensorRT 引擎时的参数设置，以及推理时的显存分配，也就是说工作前后模型参数设置需要保持一致。（指定输入输出张量的哪些维度是动态的。）

为了追求效率，ONNX 默认所有参与运算的张量都是静态的（张量的形状不发生改变）。但在实际应用中，我们又希望模型的输入张量是动态的，尤其是本来就没有形状限制的全卷积模型。因此，我们需要显式地指明输入输出张量的哪几个维度的大小是可变的。

2. 算子不兼容问题？

PyTorch 转 ONNX 时最容易出现的问题就是算子不兼容（PyTorch 算子无法翻译成 ONNX 算子），具体需要查看 [ONNX 算子文档](#)。

如果某算子确实不存在，或者算子的映射关系不满足我们的要求，我们就可能得用其他的算子绕过去，或者自定义算子了。

3. 常量折叠优化？

用已经计算好的常量节点替换都是常量输入的操作节点。



2. 多模态以 ViT 为例:

```
import clip, cv2
from PIL import Image
from clip_onnx import clip_onnx
```

1. 加载模型: (多模态任务)

```
model, preprocess = clip.load("ViT-L-14-336px.pt", device="cpu", jit=False)
```

2. 设置图像示例向量:

```
image = preprocess(Image.open("图像.png")).unsqueeze(0).cpu()
```

3. 设置文本示例向量:

```
text = clip.tokenize(["文本"]).cpu()
```

4. 转换模型:



```
onnx_model = clip_onnx(
    model, torch模型文件夹路径
    visual_path="visual.onnx", 导出图像模型文件路径
    textual_path="textual.onnx") 导出文本模型文件路径
onnx_model.convert2onnx(image, text, 模型输入
                           verbose=True) 转换信息打印
```



转换前:

 ViT-L-14-336px.pt	2022/7/22 17:49	PT 文件	912,196 KB
---	-----------------	-------	------------

转换结果:

 textual.onnx	2022/8/23 10:09	ONNX 文件	483,149 KB
 visual.onnx	2022/8/23 10:09	ONNX 文件	1,188,876 KB

分析:

1. VIT 转换完成后将文本和图像模型分别拆分出来。
2. 转换完成的两个模型体积也比原始模型大了不少, 需要进行再处理。
3. clip_onnx 底层也是调用的 torch.onnx.export 方法, 只不过封装的更方便。

3. keras 以 VGG16 为例:

```
import tf2onnx
import tensorflow as tf
from keras.models import load_model
```

1. 加载模型: (分类任务)

```
model = load_model('./weights.h5')
```

2. 设置示例向量:

```
spec = (tf.TensorSpec((None, 128, 128, 3), tf.float32, name="input"),) tuple
```



3. 转换模型:

```
model_proto, _ = tf2onnx.convert.from_keras(  
    model, keras模型文件路径  
    input_signature=spec, 模型输入（元组）  
    opset=13, 导出的ONNX版本（算子对应）  
    output_path="onnx_model.onnx" 导出模型文件路径  
)
```

转换前:

 weights.h5	2018/11/4 20:23	H5 文件	16,433 KB
--	-----------------	-------	-----------

转换后:

 onnx_model.onnx	2022/8/23 10:46	ONNX 文件	8,720 KB
---	-----------------	---------	----------

分析:

1. 转换流程整体大致相同, 需要注意的是: keras 和 tensorflow 以及 tf2onnx 版本要对应。

4. paddle 等其他框架转换流程也大致相同, 就不在过多赘述。

地址: <https://github.com/PaddlePaddle/Paddle2ONNX>

ONNX 模型推理

1. NLP 以 Bert 为例:

```
from transformers import BertForSequenceClassification, BertTokenizerFast
import onnxruntime
```

1. 初始化 tokenizer:

```
tokenizer = BertTokenizerFast(vocab_file="词表文件路径",
                             do_lower_case=True) 忽略大小写
```

2. 构建 Session:

```
sess_options = onnxruntime.SessionOptions()
```

3. 设置图优化级别:

```
sess_options.graph_optimization_level = onnxruntime.GraphOptimizationLevel.ORT_ENABLE_ALL
```

4. 构建 InferenceSession 实例:

```
session = onnxruntime.InferenceSession(onnx_model_path, sess_options,
                                       providers=["CUDAExecutionProvider",
                                                  "CPUExecutionProvider"])
```



5. 推理:

```
input_token = tokenizer(text,
                        max_length=500, truncation=True,
                        padding='max_length')
logit = self.session.run(None, {
    'input_ids': np.array([input_token['input_ids']], dtype=np.int64),
    'input_mask': np.array([input_token['attention_mask']], dtype=np.int64),
    'segment_ids': np.array([input_token['token_type_ids']], dtype=np.int64)
})
probabilities = scipy.special.softmax(logit[0], axis=-1)
```

2. 多模态以 VIT 为例:

1. 加载图像和文本向量:

```
_, preprocess = clip.load("ViT-L-14-336px.pt", device="cpu", jit=False)

image = preprocess(Image.open("cat.png")).unsqueeze(0).cpu()  [1, 3, 336, 336]
image_onnx = image.detach().cpu().numpy().astype(np.float32)

text = clip.tokenize(["a snake", "a dog", "a cat"]).cpu()  [3, 77]
text_onnx = text.detach().cpu().numpy().astype(np.int64)
```



2.提取特征向量:

```
onnx_model = clip_onnx(None)
onnx_model.load_onnx(visual_path=visual_path,
                    textual_path=textual_path,
                    logit_scale=100.0000)
onnx_model.start_sessions(providers=["CUDAExecutionProvider",
                                     "CPUExecutionProvider"])

image_features = onnx_model.encode_image(image_onnx)
text_features = onnx_model.encode_text(text_onnx)
```

3.计算对数:

```
logits_per_image, logits_per_text = onnx_model(image_onnx, text_onnx)
probs = logits_per_image.softmax(dim=-1).detach().cpu().numpy()
```

分析:

1. 提取特征向量方便做其他需求。
2. 计算分类结果的步骤中已经提取了特征向量，并对向量进行归一。
3. onnx_model 封装了计算余弦相似度的方法，总体很方便。



3. keras 以 VGG16 为例：

```
from keras.utils.image_utils import img_to_array
```

1. 提取像素矩阵并归一：

```
img = cv2.resize(cv2.imread('./2.png'), (128, 128))  
img = (np.array([img_to_array(img)], dtype='float') / 255.0).astype(np.float32)
```

2. 构建InferenceSession实例：

```
session = onnxruntime.InferenceSession(  
    "VGG16.onnx",  
    providers=["CUDAExecutionProvider"])
```

3. 获取输入实参：

```
input_name = sess.get_inputs()[0].name
```

4. 获取分类结果：

```
logits = session.run(None, {input_name: img})  
probs = scipy.special.softmax(logits[0])
```

分析：

1. 推理过程大同小异，根据需求使用一种即可。



注意:

1. 图级别优化参数选择?

ONNX Runtime 提供了各种图形优化以提高性能。图形优化本质上是图形级别的转换，从小的图形简化和节点消除到更复杂的节点融合和布局优化。

图表优化分为三个层次: Basic Extended Layout Optimizations

Basic: 保留语义的图重写，删除冗余节点和冗余计算。在图分区之前运行，因此适用于所有执行提供程序。

Extended: 复杂的节点融合。在图分区之后运行，并且仅应用于分配给 CPU 或 CUDA 执行提供程序的节点。

Layout Optimizations: 改变了适用节点的数据布局，以实现更高的性能改进。在图分区之后运行，并且仅应用于给 CPU 执行提供程序的节点。

ONNX Runtime 定义 **GraphOptimizationLevel** 枚举，以确定将启用哪个优化级别。选择一个可以实现该级别的优化，以及前面所有级别的优化。

2. 为什么不用 BertTokenizer?

BertTokenizerFast 相比 BertTokenizer 速度快，虽然它们都是最终都是继承于 PreTrainedTokenizerBase 类。



转 FP16

1. NLP 以 Bert 为例:

```
python -m onnxruntime.transformers.optimizer --input --output --float16
```

转换前:

 onnx_model.onnx	2022/8/23 17:22	ONNX 文件	399,602 KB
---	-----------------	---------	------------

转换结果:

 onnx_model.fp16.onnx	2022/8/23 17:28	ONNX 文件	210,158 KB
--	-----------------	---------	------------



2. 多模态以 VIT为例:

```
from onnx import load_model, save_model
from onnxmltools.utils.float16_converter import convert_float_to_float16
```

1.转换模型:

```
onnx_model = load_model("visual.onnx")
trans_model = convert_float_to_float16(onnx_model,
                                       keep_io_types=True) 保持FP32输入
save_model(trans_model, "visual_fp16.onnx")
```


转换前:

 textual.onnx	2022/8/23 10:09	ONNX 文件	483,149 KB
 visual.onnx	2022/8/23 10:09	ONNX 文件	1,188,876 KB

转换结果:

 textual.fp16.onnx	2022/7/25 9:42	ONNX 文件	241,633 KB
 visual.fp16.onnx	2022/7/25 9:43	ONNX 文件	594,552 KB

3. keras 以 VGG16 为例:

转换过程与 VIT 转换过程一致。

转换前:

 vgg16.onnx	2022/7/22 22:17	ONNX 文件	8,720 KB
--	-----------------	---------	----------

转换结果:

 vgg16.fp16.onnx	2022/8/23 18:11	ONNX 文件	4,379 KB
---	-----------------	---------	----------

分析:

伴随着 FP16 的转换, 模型精度有所降低, 模型体积越小, 精度降低越明显。建议每次转换完成后进行评估, 根据评估结果决策模型。

ONNX 优化

此处仅以 Bert 为例：

```
from onnxruntime_tools import optimizer
from onnxruntime_tools.transformers.onnx_model_bert import BertOptimizationOptions
```

1. 禁用嵌入层范数

```
opt_options = BertOptimizationOptions("bert")
opt_options.enable_embed_layer_norm = False
```

2. 离线融合逻辑优化

```
opt_model = optimizer.optimize_model(
    input = "onnx_model.fp16.onnx",
    model_type = "bert",
    num_heads=12,
    hidden_size=768,
    optimization_options=opt_options)
```

3. 保存模型

```
opt_model.save_model_to_file("onnx_model.opt.fp16.onnx")
```



Int8 量化

ONNX Runtime 中的量化是指 ONNX 模型的 8 位线性量化。

动态量化：动态量化只转换模型的参数类型，无需额外数据。

静态量化：需要额外的数据用于校准模型，所以相比动态量化，静态量化更加复杂一些。

一般而言推荐 RNN 系列和 transformer 系列使用动态量化，CNN 系列使用静态量化。

此处仅以 Bert 为例：

```
from onnxruntime.quantization import quantize_dynamic, QuantType
```

1. 量化为有符号 8 bit 整型

```
quantize_dynamic(  
    model_input=onnx_model_path,  
    model_output=quantized_model_path,  
    weight_type=QuantType.QInt8,  
)
```

量化结果：



分析：

1. 量化过程主要是将权重转为 Int8，提升模型推理速度。
2. 目前基于 ONNX 的量化模型只能用于 CPU 服务器，GPU 服务器无法使用。
3. 在 CPU 机器上推理速提升约为原始 pytorch 模型的 3 倍。
4. Int8量化减少了模型体积，提升了推理速度，但避免不了精度的损失。
5. 对于上述无法在 GPU 使用的问题，可以采用 tensorrt 进行 int8 量化。

TensorRT

TensorRT 是一个有助于在 NVIDIA 图形处理单元（GPU）上高性能推理 c++ 库。
在训练了神经网络之后，TensorRT 可以对网络进行压缩、优化以及运行时部署。

对于 Bert 系列模型优化的总结：

1. 可以直接采用 rbt3 进行训练（或蒸馏），模型体积约为 150 兆。
2. 训练完成的 torch 版本模型进行剪枝，在尽可能保证性能的情况下将体积缩小到 120 兆。
3. 对剪枝完成的模型进行 ONNX 转换，并进行 FP16 优化和 opt 优化，体积可缩小到 63 兆。
4. 对已完成 opt 优化的模型进行 int8 量化（建议采用 trt），体积可缩小到 30 兆。

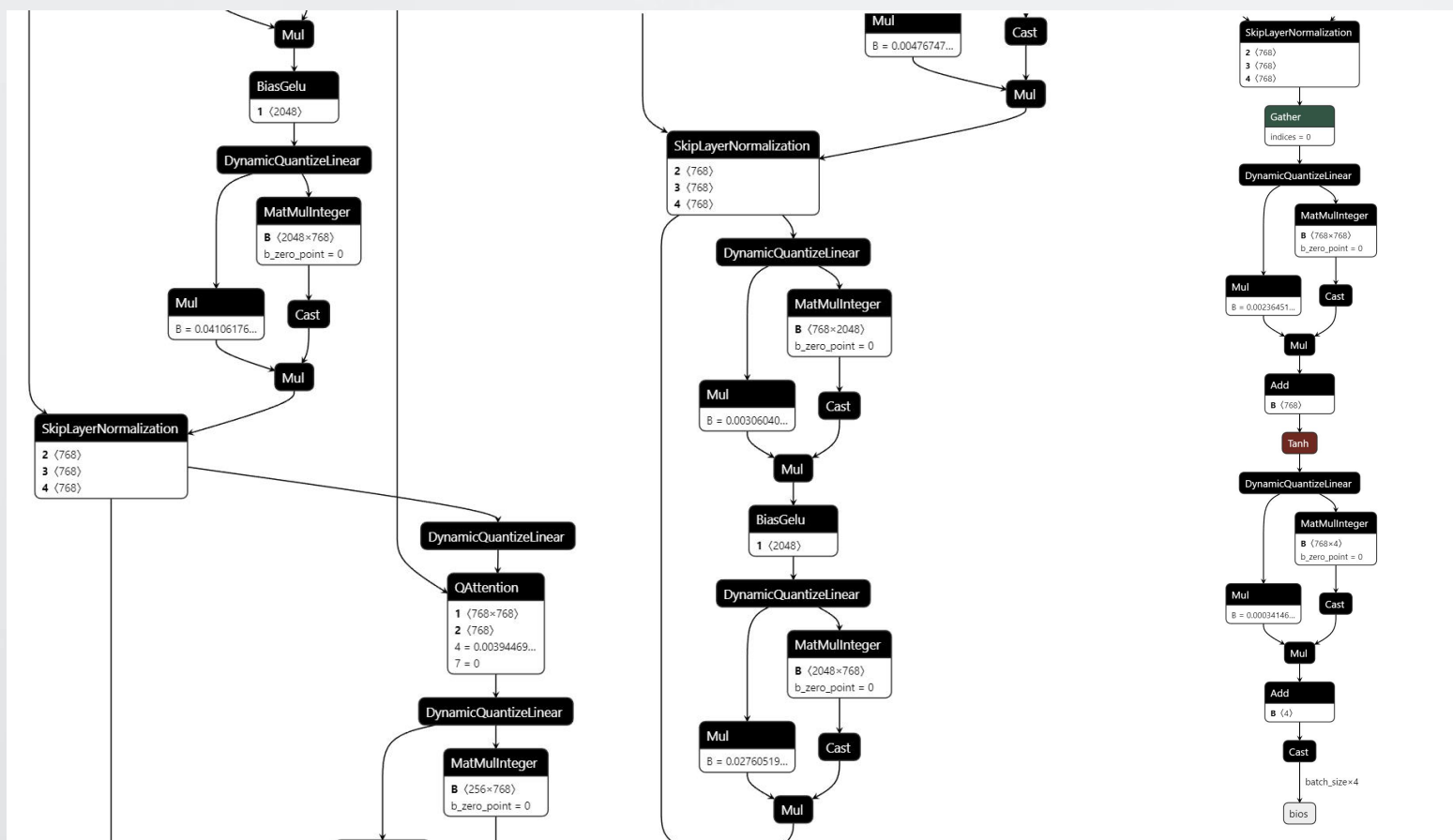
实验结果：

 pruned_H8.quant.opt.fp16.onnx	2022/8/24 14:49	ONNX 文件	30,758 KB
---	-----------------	---------	-----------

实验结论：

在精度降低了3个点的情况下，模型体积减少13倍，在 CPU 推理速度提升 15 倍。

模型可视化



安装地址: <https://github.com/lutzroeder/netron>