第五届“神威杯”国产 CPU 并行应用挑战赛

决赛赛题

赛题背景

Transformer 结构以其优秀的可并行性，可解释性，较短的位置通信路径，在自然语言处理领域，取代了传统的计算结构。BERT, GPT-2, GPT-3 等大模型在自然语言处理任务上得到了广泛的应用。近些年来，计算机视觉研究人员也关注到了 Transformer 结构的优越性，将 Transformer 应用在计算机视觉的任务中来，ViT，DETR，MoCoV3 等基于 Transformer 的网络结构在视觉任务中大放异彩。ViT 模型基于 PyTorch 实现。PyTorch 是一个开源的机器学习库，凭借其功能强大且简单易用等特性，日益成为最为流行的深度学习框架之一。PyTorch 支持多样的后端硬件，在国产申威众核 CPU 平台上，研究者们完成了对 PyTorch 的移植和优化工作（SWPyTorch）。SWPyTorch 保留了 PyTorch 的众多优秀特性，不仅为用户提供了简单易用的 Python 接口，还提供了较为完备的 C++ API。深度学习用户及框架开发者可以根据自身需求在各个层次上对 PyTorch 进行定制化扩展。例如，我们可以基于 Python 向 SWPyTorch 添加新的算子，也可以添加 C++等底层算子 kernel 并最终封装为框架 Python 前端可调用的算子。

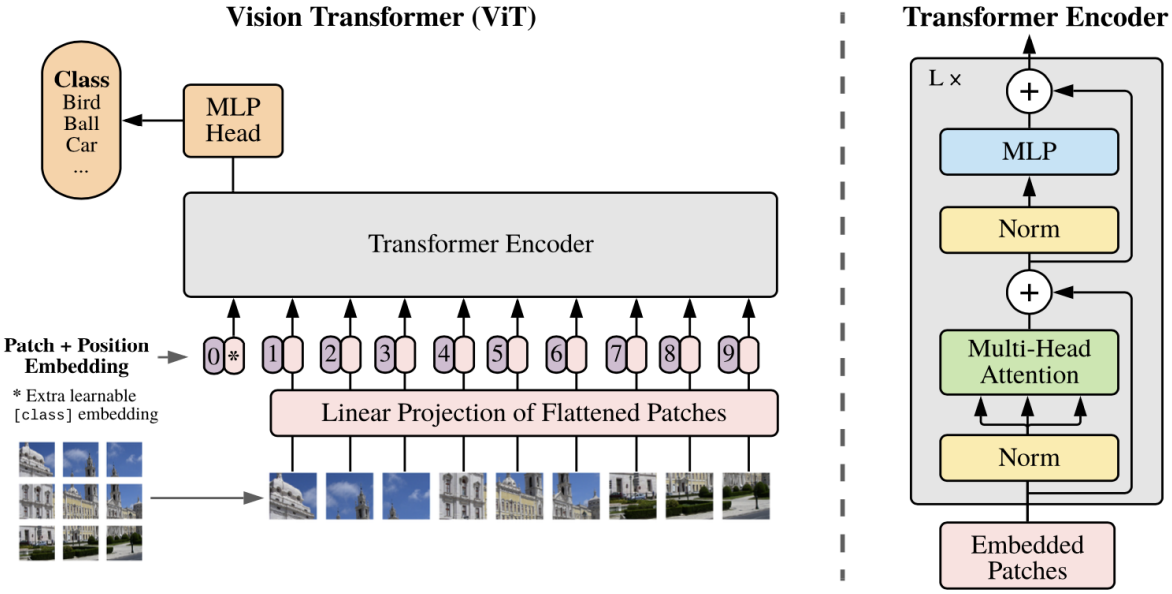
各位参赛选手在初赛中已经对 Transformer 模型中核心算子 MultiHead-Attention 的前向计算有了初步的认识，同时熟悉了国产申威处理器的架构和神威超算的环境。本次决赛，我们将在新一代神威超级计算机上完成一个基于 Transformer 的计算机视觉分类任务的模型训练。本次比赛，我们期待选手们能够基于初赛中对深度学习算子的众核优化加速经验，结合对 ViT 模型结构的充分了解和性能热点分析，充分利用 SWPyTorch 新算子开发特性，对 ViT 模型训练任务的性能进行优化。

赛题描述

新一代神威超级计算机上的 Transformer 训练任务优化。具体要求为在新一代申威众核架构下，优化基于 SWPyTorch 实现的 ViT 模型训练任务。赛题提供的代码基准为 ViT 网络

的 PyTorch 实现，底层基于申威处理器单主核运行，比赛选手需要针对其中的 Transformer 结构进行性能优化。为方便优化方案的实现，赛题提供了一套基于 CPP-Extension 扩展 SWPyTorch 的方法，即首先在 Python 层面添加扩展算子接口，然后在 C++/C 层面实现扩展算子，之后注册到 SWPyTorch 中使用。选手需要在给定的代码范围内依据自己的优化方案，自由选择目标算子进行优化，具体为通过新算子的 Python 接口替换模型(transformer.py)中的目标算子，并在 extension 模块中完成该新算子的众核加速。需要说明的是扩展 PyTorch 算子的方法有很多，为统一标准，在此我们限定选手必须使用上述给定的扩展方法完成本次赛题（算子扩展方法的具体流程和说明见相关材料）。同时需要注意，上述扩展算子的方案必须同时提供新算子的前向计算和反向计算逻辑，才能正确完成整个模型训练过程。

1. 模型结构及算法介绍



ViT 是一个使用 Transformer 结构进行图像分类任务的网络结构。先将图像划分为固定尺寸大小的 patches，然后对每一个 patch 进行 linearly embed，并添加 position embedding，再将结果输入至一个 transformer encoder 结构中。为了进行分类任务，在进行 linearly embed 之后，额外添加了一个可学习的 classification token。将 transformer encoder 输入的 classification token，经过一个 MLP Head 进行分类。

例如在我们使用的 MNIST 分类数据集中，每一张图片的尺寸大小为(28,28)，维度为 1，共有 10 类。将 patches 的尺寸设置为 4，即共有 49 个 patch，每一个 batch 的图

片 linearly embed 之后，shape 为(batch_size, 49, dim)。然后加上 classification token，则 shape 为(batch_size, 50, dim)。即输入 transformer encoder 的 tensor shape 为(batch_size, 50, dim)，输出为(batch_size,50,dim)，我们将输出的 shape 为(batch_size, dim)的 classification token 作为分类的特征，经过一个 MLP Head 进行 10 分类。

Transformer Encoder 由 Multi-Head Attention 和 FeedForward 组成。初赛中我们已经很熟悉 Multi-Head Attention 结构，复赛中我们使用的版本与初赛完全相同，（除了将初赛中为了降低难度采用的简化归一化操作换为正式的 softmax 操作）。FeedForward 简单来说就是两层的 MLP(多层感知机)。参赛选手可重点关注 Transformer Encoder 的计算。

2. 赛题代码结构说明

赛题提供 ViT 模型实现和 CPP Extension 两个模块的代码，比赛选手需要在赛题给定允许修改的代码范围内实现优化方案。

- ViT 模型:：位于 model/目录

文件名	说明	范围
train.py	训练任务主逻辑	不允许修改
vit.py	ViT 网络文件	
config.py	超参数配置文件	
metric.py	分类任务 metric	
transformer.py	Transformer 模块文件	允许修改
swnn.py	扩展算子 Python 接口代码	允许修改

- extension 模块：位于 sw_extension/目录

文件名	说明	范围

swextension.cpp	扩展算子 C++接口代码	允许修改
swops.h	扩展算子头文件	允许修改
args.h	拓展算子参数结构体	允许修改
master.c	扩展算子主核实现	允许修改
slave.c	扩展算子从核实现	允许修改

不允许增加新文件以及修改其他不在上述范围内的文件。

3. 赛题评测数据规模描述

赛题分为两类 Case： Case A 在单节点上训练；Case B 在最多 256 个节点上训练。

测例	要求	参数范围
Case A	单节点训练	BatchSize：16 的倍数，不超过 1024 ViT 网络参数范围： After embedding dim： 32 的倍数，不超过 2048 Depth of MultiHead : 2-8 Heads number： 2-6 Feed Forward hidden layer dim：32 的倍数，不超过 2048
Case B	多节点执行	赛程第 3 周发布

评分规则

1. 比赛将使用组委会提供的八组测试样例进行测试，其中 Case A 为 5 组，Case B 为 3 组。每组测例 10 分，共 80 分。评分所用的测试参数与题目中提供的测试参数均不相同；对于 Case A 的每组测试参数，完成 10 个 Epoch 的训练，统计运行时间，按照运行时间

进行排名计算得分，运行最快者得 10 分。

2. 选手必须保证计算结果的正确性，本次比赛采用的双重验证方法，验证 1:验证 Python 层 MultiheadAttention 和 FeedForward 算子的正向和反向计算的正确性，采用绝对误差对浮点数正确性进行判断，误差限为 10^-4；验证 2:训练完指定数量的 Epoch，验证集上最好的准确率超过 90% ;任意一组测试参数下不满足验证条件，本次提交成绩无效。
3. 选手成绩取个人多次提交里面的最好成绩。
4. 如果时间一样，先提交的选手排名靠前。
5. 选手优化可以从如何切分输入张量并尽可能重用数据的角度考虑，针对不同的参数范围，其最高效的划分方式可能不同。选手着重提高代码并行计算效率和发挥申威 CPU 计算潜力，参赛者若修改源码中已明确不能修改部分或做其他不合理的改动将导致成绩无效。
6. 编译选项，编译链接选项位于 Makefile 文件中，不得更改。

相关材料

1. SWPyTorch 相关代码介绍与说明

由于SWPyTorch代码结构本身庞大且复杂，在本次比赛中，我们规定**不可对SWPyTorch的代码进行修改**。如需提供优化版本的算子/融合算子实现，选手只能利用 PyTorch 的 C++ Extension 接口以 Out-of-tree 的形式进行开发。

下面介绍如何通过 PyTorch C++ Extension 接口向 PyTorch 添加算子实现。

2. 向 SWPyTorch 添加算子

该目录给出了通过 CPP-Extension 的方式向 PyTorch 注册添加新算子的示例。

i. 示例代码位置：sw_extension

ii. 扩展算子的具体步骤为：

在 swextension.cpp 中编写新算子的前向和反向计算函数接口，选手可以参照 demo 代码及 PyTorch 官方 tutorial 及 API reference（见文末链接）

```cpp
std::vector<torch::Tensor> swlinear_backward(
    torch::Tensor grad_output,
    torch::Tensor input,
    torch::Tensor weight) {

  auto N = weight.size(0); // output features
  auto K = weight.size(1); // input features
  auto M = input.numel() / K; // batch

  // dx = dy * W
  auto d_input = torch::zeros_like(input);
  // dw = dyT * X
  auto d_weight = torch::zeros_like(weight);

  // get raw data pointer
  auto dx = d_input.data_ptr();
  auto dw = d_weight.data_ptr();
  auto dy = grad_output.data_ptr();
  auto x = input.data_ptr();
  auto w = weight.data_ptr();

  // call op kernel impl
  swptex_mm(dy, x, dw, N, K, M, 1, 0);
  swptex_mm(dy, w, dx, M, K, N, 0, 0);

  return {d_input, d_weight};
}
```

并通过 pybind11 绑定到 Python 中

```cpp
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
  m.def("swlinear_forward", &swlinear_forward, "swLinear forward");
  m.def("swlinear_backward", &swlinear_backward, "swLinear backward");
}
```

在 swops 中编写神威平台算子众核实现

```
extern SLAVE_FUN(sw_slave_mm_AB)(swptex_mmPara_t);
extern SLAVE_FUN(sw_slave_mm_ATB)(swptex_mmPara_t);
extern SLAVE_FUN(sw_slave_mm_ABT)(swptex_mmPara_t);

extern __cross void* para_cross; // param on cross seg

int swptex_mm(const void* A, const void *B, void *C, size_t M,
              size_t N, size_t K, int transposeA, int transposeB){
    swptex_mmPara para;
    para.A = A; para.B = B; para.C = C;
    para.M = M; para.N = N; para.K = K;
    para_cross = &para; // cross seg variable to pass param
    athread_init_cgs();
    if(!transposeA && transposeB){
        athread_spawn_cgs(sw_slave_mm_ABT, &para);
    }else if(transposeA && !transposeB){
        athread_spawn_cgs(sw_slave_mm_ATB, &para);
    }else if(!transposeA && !transposeB){
        athread_spawn_cgs(sw_slave_mm_AB, &para);
    }else{
        printf("not supported\n");
        return 0;
    }
    athread_join_cgs();
}
```

在 swops 中执行 make 将算子众核实现编译为静态库文件。

在 sw_extension 文件执行 install.sh 提交作业， 编译算子扩展并安装为 Python 包，至此已经可以在 python 中 import swextension 模块。

参照 demo 示例，编写 swnn.py，将算子进一步封装为 nn.Module

```python
from torch import nn
from torch.autograd import Function
import torch
import swextension

class swLinearFunction(Function):
    @staticmethod
    def forward(ctx, input, weight):
        outputs = swextension.swlinear_forward(input, weight)
        variables = [input] + [weight]
        ctx.save_for_backward(*variables)
        return outputs[0]

    @staticmethod
    def backward(ctx, grad_output):
        d_input, d_weight = swextension.swlinear_backward(grad_output,
*ctx.saved_variables)
        return d_input, d_weight

class swLinear(nn.Module):
    def forward(self, input):
        return swLinearFunction.apply(input, self.weight)
```

最终在 transformer.py 中通过 from swnn import swLinear 调用实现的模块

```python
from torch import nn
from swnn import swLinear

class Attention(nn.Module):
    def __init__(self, dim, heads = 8, dim_head = 64,):
        super().__init__()
        inner_dim = dim_head *  heads
        #self.to_qkv = nn.Linear(dim, inner_dim * 3, bias = False)
        self.to_qkv = swLinear(dim, inner_dim * 3)
```

3. 反向求导：深度学习框架如 PyTorch 提供了自动求导的机制，因此，梯度计算的逻辑未在代码中显示表达；为方便选手理解神经网络训练过程中的反向计算逻辑，我们提供了一份 Transformer 层 backward 的主核 C++实现以供参考。

官方 tutorial reference 链接：https://pytorch.org/tutorials/advanced/cpp_extension.html

官方 API reference 链接: https://pytorch.org/cppdocs/api/classat_1_1_tensor.html