Deep Learning Project: Differentiable Neural Architecture Search and Quantization-Aware Training using PLiNIO

Tiziano Tezze VR504577

Artificial Intelligence Master's Degree - A.Y. 2023/2024

Abstract

Developing accurate *Deep Neural Networks* (**DNNs**), which are able to efficiently run in applications whose execution is constrained on edge devices, is an high demanded yet difficult task. Finding such DNNs requires optimization pipelines able to extensively explore the huge space of hyperparameter combinations. In this environment, we can find both *Differentiable Neural Architecture Search* (**DNAS**) and *Quantization-Aware Training* (**QAT**) methods, which are able to carry out this task in a gradient-descent way. Hence, these methods are able to perform this kind of searching process during training.

PLiNIO (**Pl**ung-and-play **Li**ghtweight **N**eural **I**nference **O**ptimization) is an open-source library implementing a comprehensive state-of-the-art DNN design optimization techniques based on gradient-based optimization. This project purpose is to show how these methods provided by this unified framework can be easily implemented in the training process of a general architecture. More specifically, the target networks of the optimization process were three instances of VGG16 *Convolutional Neural Network* (**CNN**) trained on 3 different datasets: MNIST, FashionMNIST, NotMNIST.

In the following sections, we will briefly introduce the optimization methods provided by PLiNIO for then going through both the setup and the results of our experiments. Lastly, there will be a section dedicated to limitations of the library, arose during this tryout.

I. Introduction

The main optimization methods provided by the library are: **PIT** (*Pruning In Time*), **MPS** (*Mixed Precision Search*) and **SuperNet**. The general way to turn a standard training PyTorch loop into a PLiNIO optimization is given by the follwing code snippet:

```
1  from torchvision.models import vgg16
2  from plinio.methods import Method
3  from plinio.cost import Cost
4  
5  # Define the model to optimize
6  model = vgg16()
7  
8  # Convert to PLiNIO object
9  model = Method(model, {'cost': Cost})
10 
11 # TRAINING LOOP
12 for epoch in range(N_EPOCHS):
13     for sample, target in data:
14         # Make predictions
15         output = model(sample)
16         # Compute the loss
17         loss = loss_fn(output, target)
18 
19         # Add model's cost to the loss
20         loss += model.get_cost('cost')
21 
22         # Update model's parameters
23         optimizer.zero_grad()
24         loss.backward()
25         optimizer.step
26 
27 # Convert the model back to PyTorch
28 exported_model = model.export()
```

where `Method` (lines 2 and 9) is a placeholder for one of the provided optimization methods and `Cost` (lines 3 and 9) is a placeholder for one (or more) of the provided cost metrics. An important thing to point out is that the model we choose to optimize (in our case VGG16) must be a subclass of `nn.Module`.

The conversion of the PyTorch model to a PLiNIO object (line 9) slightly changes depending on the chosen `Method`. The arguments listed in the snippet are shared among every available method. However, more arguments may be required depending on the chosen optimization technique.

1) PIT: This DNAS method implements the general concept of **Pruning**, which aims to compress the size of the model by removing either connections between neurons (weights) or entire neurons from the network. *Weight-based* pruning is more popular, since it easier to implement without hurting the performance of the network, which retains its original structure.
In PLiNIO, the method `PIT` implements weight-based pruning, and applies it as the model is trained to all its **Convolutional** and **Fully Connected** layers (current supported layers by the framework). Connections in the network are pruned according to which PLiNIO `Cost` metrics we consider and how important they are in the computation of the loss. Currently, the general cost metrics available for PIT optimization are:

- `params`, which counts the number of weights of the supported layers, ignoring biases;
- `ops`, which counts the number of *FLOPS* (Floating-point Operations) per inference performed by the supported layers.

For example, suppose that we want to convert our instance of VGG16 to a PLiNIO object, which carries out PIT optimization according to the number of weights of the network, this is the code snippet:

```
1  from torchvision.models import vgg16
2  from plinio.methods import PIT
3  from plinio.cost import params
4
5  # Standard model
6  model = vgg16()
7
8  # PLiNIO object
9  model = PIT(
10     model=model,
11     input_shape=input_shape,
12     cost={'weights': params}
13 )
```

where `input_shape` (line 8) refers to the shape of the tensor of the input data, without considering the batch size. And then, within the training loop, as we compute the loss:

```
# Standard loss
loss = loss_fn(output, target)

# Loss + PLiNIO cost
loss += strength*model.get_cost('
weights')
```

where `stength` is an hyper-parameter stating how much we take the cost into account in the computation of the loss.

2) SuperNet: This is a **path-based** DNAS method, whose execution arounds the concept of *supernet*. The supernet is built by replacing each layer (or module) of a standard DNN with a set of alternatives. Then, during training, the best alternative is chosen again according to which PLiNIO `Cost` metrics we consider and how important they are in the computation of the loss. In PLiNIO the method `SuperNet` shares the same cost metrics as PIT. From the official GitHub page (PLiNIO SuperNet), the following example turns a simple standard DNN into a PLiNIO object for SuperNet optimization:

1. first we need to define the network with layers suitable for SuperNet optimization:

```
from torch import nn
from plinio.methods.supernet import
    SuperNetModule

class MyNN(nn.Module):
    def __init__(self):
        self.conv = SuperNetModule([
            # Alternative 1
            nn.Conv2d(
                32, 32, 3,
                padding='same'
            ),
            # Alternative 2
            nn.Sequential(
                nn.Conv2d(
                    32, 32, 3,
                    padding='same'
                ),
                nn.Conv2d(
                    32, 32, 3,
                    padding='same'
                ),
            ),
            # Alternative 3
           nn.Conv2d(
                32, 32, 5,
                padding='same'
            ),
            # Alternative 4
            nn.Identity()
        ])

    def forward(self, x):
        return self.conv(x)
```

by using PLiNIO `SuperNetModule`, in which we list the alternatives for that specific layer (or module). Notice that the alternatives must have the same shape of the output tensor.

2. lastly we convert the network into a PLiNIO object carrying out SuperNet optimization (assuming `params` as cost metric):

```python
from plinio.methods import SuperNet
from plinio.cost import params

# Standard model
model = MyNN()

# PLiNIO object
model = SuperNet(
    model=model,
    input_shape=input_shape,
    cost={'weights': params}
)
```

The computation of the loss during training is standard to the one of PIT.

3) MPS: **Quantization** is a DNN optimization technique constisting in the approximation of both floating point weights and floating point activations by using low bitwidth integers in order to improve both model size and efficiency. That is the network is optimized by lowering the amount of bits (known as *precision*) required to store weights and activations. In this field, *Quantization-Aware Training* is a set of techniques, which aim to simulate the effect of quantization at training time, while the actual conversion can be performed post-training once the optimized architecture has been found (this fact will be pointed out as results of the experiments will be discussed).
Since different layers might require different precision levels, *Mixed Precision Search* is the QAT method, which has been implemented in PLiNIO. It works by quantizing each FC and Conv layer's weights and activations at different precision levels at the same time. As training goes on, MPS selects for each one of these layers the best bitwidth to assign according to both the selected PLiNIO `Cost` metrics and how much they affect the loss function. Currently, MPS has its own set of metrics and the general ones are:

- `params_bit`, which counts the umber of bits required to store the weights of the network;
- `ops_bit`, which counts the number of *bit-operations* (**bitops**) defined as:

  $$\sum_{b_w, b_x} b_w \cdot b_x \cdot OPS_{b_w, b_x}$$

  where b_w and b_x are the possible weights and activations bitwidths and OPS_{b_w, b_x} are the number of FLOPS performed with those bitwidths.

For example, suppose that we want to convert our instance of VGG16 to a PLiNIO object carrying out MPS considering 8 bits and 16 bits as possible precision levels according to the number of bits required to store the weights of the network, this is the code snippet:

```python
1  from torchvision.models import vgg16
2  from plinio.methods.mps import MPS,
       MPSType, get_default_qinfo
3  from plinio.cost import params_bit
4
5  # Standard model
6  model = vgg16()
7
8  # PLiNIO object
9  model = MPS(
10     model=model,
11     qinfo=get_default_qinfo(
12         w_precision=(8,16),
13         a_precision=(8,16)
14     )
15     w_search_type=MPSType.PER_LAYER,
16     cost={'weights': params_bit}
17 )
```

where `get_default_qinfo` (line 2 and 11) is the method allowing the listing of the possible bitwidths to consider during optimization. At line 15, `w_search_type` defines the way with which MPS optimzation is carried out. Currently, PLiNIO offers two different approaches:

1. `PER_LAYER` is the default approach, which assigns the same precision level to entire weight and activation tensors of each layer;
2. `PER_CHANNEL` is an approach supporting only weights, which assigns different precision levels to each output channel of the weight tensor of a convolutional layer.

However, the framework currently crashes when exporting model optimized with the `PER_CHANNEL` approach.

II. Experiments Setup and Results

As mentioned in the abstract, the target networks of the experiments were 3 instances of VGG16 CNN trained on 3 different datasets: MNIST, NotMNIST and FashionMNIST. These datasets consist in 224×244 grayscale images, each of which belongs to one of 10 classes. As train/test split, for each dataset, 60'000 images have been used for training, while 10'000 images have been used for testing. All

Model	MNIST	FashionMNIST	NotMNIST
VGG16	54m 43s	54m 33s	54m 32s
PIT	59m 55s	58m 37s	58m 37s
PIT*	1h 09m 32s	1h 08m 32s	1h 10m 20s
MPS	4h 55m 47s	5h 05m 26s	5h 19h 33s
SuperNet	8h 17m 00s	8h 34m 43s	8h 21m 15s
SuperNet*	6h 49m 31s	9h 00m 03s	8h 46m 33s

Table 1: Required times for training

Model	**Model Size** (MB)	**Accuracy** (%)	**Inference** (ms)
VGG16	512.32	88.35	5.000
PIT	112.1	86.87	5.408
PIT*	112.1	87.30	5.237
MPS	-	89.39	141.856
SuperNet	136.21	88.76	5.321
SuperNet*	136.21	88.68	5.273

Table 2: Test results on FashionMNIST

instances of VGG16 have been trained through 5 epochs of *Stochastic Gradient Descent* (**SGD**) with a starting learning rate of 0.001 and a batch size of 16. Both training and testing have been performed on an 8GB graphics card NVIDIA GeForce RTX 4060ti.

III. Considerations

As we can see in Table 1, all gradient based optimization methods required more time for training than standard VGG16. This is very noticeable both for MPS and SuperNet optimization techniques, since the former feeds the data through all the possible bit-widths and the latter feeds the data through all possible paths (alternative modules) we have defined.

In Table 2, we can see how both DNAS methods succeeded in finding a model much lighter than standard VGG16:

- PIT found a model 78.12% lighter than the baseline at the cost of just 1.48% in terms of accuracy;
- SuperNet decided for a network which is 73.41% lighter than the baseline with no cost in terms of accuracy.

In the same table, we can also see version of the former techniques (denoted with *), which used a specific regularizer called **DUCCIO**. This feature provided by PLiNIO allows cost metrics to be treated as constraints rather than just secondary objectives. It works by penalizing the loss whenever these constraints are exceeded. This trait forces the optimization techniques to search only in a restricted space of networks. Both variants trained with DUCCIO achieved the same model of the optimization technique following the standard pipeline.

IV. Limitations

During the tryout of this framework, important limitations and issues arose probably due to its youth.
Looking at Table 2, notice that MPS misses the value under the "**Model Size**" column. This is due to the fact that `model.export()` does not export the model with modified bit-widths. Rather, it exports the model adding to each layer quantizers for both weights and activations that quantize data according to the learnt bit-widths. This explains also why the average inference time is much higher than all the other models.
Another issue concerned models trained with SuperNet. At the end of the training phase, exported models retained the structure achieved by the optimization technique, but weights were not updated. This means that models required further training in order to achieve the results of Table 2 in terms of accuracy.