QPyTorch Documentation

Release 0.0.1 alpha

Tianyi Zhang, Zhiqiu Lin, Christopher De Sa

Jan 14, 2022

Guides and Tutorials:

CHAPTER 1

QPyTorch Functionality Overview

1.1 Introduction

In this notebook, we provide an overview of the major features of QPyTorch.

[1]:
```
import torch
import qtorch
```

1.2 Quantization

QPyTorch supports three different number formats: fixed point, block floating point, and floating point.

QPyTorch provides quantization functions that quantizes pytorch tensor.

[2]:
```
from qtorch.quant import fixed_point_quantize, block_quantize, float_quantize
```

[3]:
```
full_precision_tensor = torch.rand(5)
print("Full Precision: {}".format(full_precision_tensor))
low_precision_tensor = float_quantize(full_precision_tensor, exp=5, man=2, rounding=
↪"nearest")
print("Low Precision: {}".format(low_precision_tensor))
```

```
Full Precision: tensor([0.1241, 0.3602, 0.7104, 0.8344, 0.0211])
Low Precision: tensor([0.1250, 0.3750, 0.7500, 0.8750, 0.0195])
```

QPyTorch supports both nearest rounding and stochastic rounding.

[4]:
```
nearest_rounded = float_quantize(full_precision_tensor, exp=5, man=2, rounding=
↪"nearest")
stochastic_rounded = float_quantize(full_precision_tensor, exp=5, man=2, rounding=
↪"stochastic")
```

(continues on next page)

(continued from previous page)

```
print("Nearest: {}".format(nearest_rounded))
print("Stochastic: {}".format(stochastic_rounded))
```

```
Nearest: tensor([0.1250, 0.3750, 0.7500, 0.8750, 0.0195])
Stochastic: tensor([0.1250, 0.3750, 0.7500, 0.8750, 0.0195])
```

1.3 Autograd

QPyTorch offers a pytorch nn.Module wrapper to integrate quantization into auto differention. A Quantizer module can use different low-precision number formats for forward and backward propagation.

[5]:
```
# First define number formats used in forward and backward quantization
from qtorch import FixedPoint, FloatingPoint
forward_num = FixedPoint(wl=4, fl=2)
backward_num = FloatingPoint(exp=5, man=2)

# Create a quantizer
from qtorch.quant import Quantizer
Q = Quantizer(forward_number=forward_num, backward_number=backward_num,
              forward_rounding="nearest", backward_rounding="stochastic")
```

[6]:
```
# Use QPyTorch Quantizer just as any other nn.Modules
from torch.nn import Module, Linear
class LinearLP(Module):
    """
    a low precision Logistic Regression model
    """
    def __init__(self):
        super(LinearLP, self).__init__()
        self.W = Linear(5, 1)

    def forward(self, x):
        out = self.W(x)
        out = Q(out)
        return out

lp_model = LinearLP()
```

[7]:
```
# forward low precision model, get low precision output
fake_input = torch.rand(5)
lp_output = lp_model(fake_input)
print("Low Precision Output: {}".format(lp_output))
```

```
Low Precision Output: tensor([-0.2500], grad_fn=<RoundingBackward>)
```

[8]:
```
# backward propagation is quantized automatically
from torch import sigmoid
from torch.nn import BCELoss
lp_model.zero_grad()
criterion = BCELoss()
label = torch.Tensor([0])
loss = criterion(sigmoid(lp_output), label)
loss.backward()
```

1.4 Low Precision Optimization

1.4.1 Weight and Gradient Quantization

In the previous example, the forward and backward signals are quantized into low precision. However, if we optimize our model using gradient descent, the weight and gradient may not necessarily be low precision. QPyTorch offers a low precision wrapper for pytorch optimizers and abstracts the quantization of weights, gradients, and the momentum velocity vectors.

[9]:
```
from torch.optim import SGD
from qtorch.optim import OptimLP

optimizer = SGD(lp_model.parameters(), momentum=0.9, lr=0.1) # use your favorite␣
↪optimizer
# define custom quantization functions for different numbers
weight_quant = lambda x : float_quantize(x, exp=5, man=2, rounding="nearest")
gradient_quant = lambda x : float_quantize(x, exp=5, man=2, rounding="nearest")
momentum_quant = lambda x : float_quantize(x, exp=6, man=9, rounding="nearest")
# turn your optimizer into a low precision optimizer
optimizer = OptimLP(optimizer,
                    weight_quant=weight_quant,
                    grad_quant=gradient_quant,
                    momentum_quant=momentum_quant)
```

[10]:
```
print("Weight before optimizer stepping: \n{}".format(lp_model.W.weight.data))
print("Gradient before optimizer stepping: \n{}\n".format(lp_model.W.weight.grad))
optimizer.step()
print("Weight after optimizer stepping: \n{}".format(lp_model.W.weight.data))
print("Gradient after optimizer stepping: \n{}".format(lp_model.W.weight.grad))
optimizer.zero_grad() #
```

```
Weight before optimizer stepping:
tensor([[-0.1850,  0.1250, -0.1007, -0.0862,  0.3034]])
Gradient before optimizer stepping:
tensor([[0.1051, 0.2755, 0.0375, 0.1643, 0.1883]])

Weight after optimizer stepping:
tensor([[-0.1875,  0.0938, -0.1094, -0.1094,  0.3125]])
Gradient after optimizer stepping:
tensor([[0.1094, 0.2500, 0.0391, 0.1562, 0.1875]])
```

1.4.2 Gradient Accumulator

One popular practice in low precision training is to utilize a higher precision gradient accumulator. The gradients, after multiplied with learning rate, modified by the momentum terms, are added onto the high precision gradient accumulator. Upon next iteration of forward and backward propagation, the weights are re-quantized from the gradient accumulator so expensive computations are still done in low precision.

QPyTorch integrates this process into the low precision optimizer.

[11]:
```
# Let's quickly repeat the above example
lp_model = LinearLP()
fake_input = torch.rand(5)
lp_output = lp_model(fake_input)
print("Low Precision Output: {}".format(lp_output))
```

(continues on next page)

(continued from previous page)

```
lp_model.zero_grad()
criterion = BCELoss()
label = torch.Tensor([0])
loss = criterion(sigmoid(lp_output), label)
loss.backward()
```

```
Low Precision Output: tensor([0.2500], grad_fn=<RoundingBackward>)
```

[12]:
```
# define a low precision optimizer with gradient accumulators
optimizer = SGD(lp_model.parameters(), momentum=0, lr=0.1)
weight_quant = lambda x : float_quantize(x, exp=5, man=2, rounding="nearest")
gradient_quant = lambda x : float_quantize(x, exp=5, man=2, rounding="nearest")
acc_quant = lambda x : float_quantize(x, exp=6, man=9, rounding="nearest") # use␣
↪higher precision for accumulator
optimizer = OptimLP(optimizer,
                    weight_quant=weight_quant,
                    grad_quant=gradient_quant,
                    momentum_quant=momentum_quant,
                    acc_quant=acc_quant)
```

[13]:
```
print("Weight before optimizer stepping: \n{}\n".format(lp_model.W.weight.data))
optimizer.step()
print("after stepping, high precision accumulator : \n{}".format(optimizer.weight_
↪acc[lp_model.W.weight]))
print("after stepping, low precision weight : \n{}".format(lp_model.W.weight.data))
optimizer.zero_grad()
```

```
Weight before optimizer stepping:
tensor([[ 0.2943, -0.1296, -0.4130, -0.2599, -0.4059]])

after stepping, high precision accumulator :
tensor([[ 0.2817, -0.1309, -0.4253, -0.2603, -0.4185]])
after stepping, low precision weight :
tensor([[ 0.3125, -0.1250, -0.4375, -0.2500, -0.4375]])
```

1.5 High-level Helper

QPytorch also provide a useful helper that automatically turn a predefined pytorch model into a low-precision one.

[14]:
```
from qtorch.auto_low import sequential_lower
class LinearFP(Module):
    """
    a low precision Logistic Regression model
    """
    def __init__(self):
        super(LinearFP, self).__init__()
        self.W = Linear(5, 1)

    def forward(self, x):
        out = self.W(x)
        return out

fp_model = LinearFP()
```

(continues on next page)

(continued from previous page)

```
forward_num = FixedPoint(wl=4, fl=2)
backward_num = FloatingPoint(exp=5, man=2)
lp_model = sequential_lower(fp_model, layer_types=['linear'],
                            forward_number=forward_num, backward_number=backward_num)
```

[15]:
```
print("Full Precision Model: ")
print(fp_model)
```

```
Full Precision Model:
LinearFP(
  (W): Linear(in_features=5, out_features=1, bias=True)
)
```

[16]:
```
print("Low Precision Model: ")
lp_model
```

```
Low Precision Model:
```

[16]:
```
LinearFP(
  (W): Sequential(
    (0): Linear(in_features=5, out_features=1, bias=True)
    (1): Quantizer()
  )
)
```

CHAPTER 2

CIFAR10 Low Precision Training Example

In this notebook, we present a quick example of how to simulate training a deep neural network in low precision with QPyTorch.

[1]:
```
# import useful modules
import argparse
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from qtorch.quant import Quantizer, quantizer
from qtorch.optim import OptimLP
from torch.optim import SGD
from qtorch import FloatingPoint
from tqdm import tqdm
import math
```

We first load the data. In this example, we will experiment with CIFAR10.

[2]:
```
# loading data
ds = torchvision.datasets.CIFAR10
path = os.path.join("./data", "CIFAR10")
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
train_set = ds(path, train=True, download=True, transform=transform_train)
```

(continues on next page)

(continued from previous page)

```
test_set = ds(path, train=False, download=True, transform=transform_test)
loaders = {
        'train': torch.utils.data.DataLoader(
            train_set,
            batch_size=128,
            shuffle=True,
            num_workers=4,
            pin_memory=True
        ),
        'test': torch.utils.data.DataLoader(
            test_set,
            batch_size=128,
            num_workers=4,
            pin_memory=True
        )
}
```

```
Files already downloaded and verified
Files already downloaded and verified
```

We then define the quantization setting we are going to use. In particular, here we follow the setting reported in the paper "Training Deep Neural Networks with 8-bit Floating Point Numbers", where the authors propose to use specialized 8-bit and 16-bit floating point format.

[3]:
```
# define two floating point formats
bit_8 = FloatingPoint(exp=5, man=2)
bit_16 = FloatingPoint(exp=6, man=9)

# define quantization functions
weight_quant = quantizer(forward_number=bit_8,
                         forward_rounding="nearest")
grad_quant = quantizer(forward_number=bit_8,
                         forward_rounding="nearest")
momentum_quant = quantizer(forward_number=bit_16,
                         forward_rounding="stochastic")
acc_quant = quantizer(forward_number=bit_16,
                         forward_rounding="stochastic")

# define a lambda function so that the Quantizer module can be duplicated easily
act_error_quant = lambda : Quantizer(forward_number=bit_8, backward_number=bit_8,
                         forward_rounding="nearest", backward_rounding="nearest")
```

Next, we define a low-precision ResNet. In the definition, we recursively insert quantization module after every convolution layer. Note that the quantization of weight, gradient, momentum, and gradient accumulator are not handled here.

[4]:
```
def conv3x3(in_planes, out_planes, stride=1):
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
                     padding=1, bias=False)

class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, quant, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.bn1 = nn.BatchNorm2d(inplanes)
        self.relu = nn.ReLU(inplace=True)
```

(continues on next page)

(continued from previous page)

```
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv2 = conv3x3(planes, planes)
        self.downsample = downsample
        self.stride = stride
        self.quant = quant()

    def forward(self, x):
        residual = x

        out = self.bn1(x)
        out = self.relu(out)
        out = self.quant(out)
        out = self.conv1(out)
        out = self.quant(out)

        out = self.bn2(out)
        out = self.relu(out)
        out = self.quant(out)
        out = self.conv2(out)
        out = self.quant(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual

        return out

class PreResNet(nn.Module):

    def __init__(self,quant, num_classes=10, depth=20):

        super(PreResNet, self).__init__()
        assert (depth - 2) % 6 == 0, 'depth should be 6n+2'
        n = (depth - 2) // 6

        block = BasicBlock

        self.inplanes = 16
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1,
                               bias=False)
        self.layer1 = self._make_layer(block, 16, n, quant)
        self.layer2 = self._make_layer(block, 32, n, quant, stride=2)
        self.layer3 = self._make_layer(block, 64, n, quant, stride=2)
        self.bn = nn.BatchNorm2d(64 * block.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.avgpool = nn.AvgPool2d(8)
        self.fc = nn.Linear(64 * block.expansion, num_classes)
        self.quant = quant()
        IBM_half = FloatingPoint(exp=6, man=9)
        self.quant_half = Quantizer(IBM_half, IBM_half, "nearest", "nearest")
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
            elif isinstance(m, nn.BatchNorm2d):
```

(continues on next page)

(continued from previous page)

```
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def _make_layer(self, block, planes, blocks, quant, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, planes * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
            )

        layers = list()
        layers.append(block(self.inplanes, planes, quant , stride, downsample))
        self.inplanes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes, quant))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.quant_half(x)
        x = self.conv1(x)
        x = self.quant(x)

        x = self.layer1(x)  # 32x32
        x = self.layer2(x)  # 16x16
        x = self.layer3(x)  # 8x8
        x = self.bn(x)
        x = self.relu(x)
        x = self.quant(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        x = self.quant_half(x)

        return x
```

[5]:
```
model = PreResNet(act_error_quant)
```

[7]:
```
device = 'cuda' # change device to 'cpu' if you want to run this example on cpu
model = model.to(device=device)
```

We now use the low-precision optimizer wrapper to help define the quantization of weight, gradient, momentum, and gradient accumulator.

[8]:
```
optimizer = SGD(model.parameters(), lr=0.05, momentum=0.9, weight_decay=5e-4)
optimizer = OptimLP(optimizer,
                    weight_quant=weight_quant,
                    grad_quant=grad_quant,
                    momentum_quant=momentum_quant,
                    acc_quant=acc_quant,
                    grad_scaling=1/1000 # do loss scaling
)
```

We can reuse common training scripts without any extra codes to handle quantization.

[9]:
```
def run_epoch(loader, model, criterion, optimizer=None, phase="train"):
    assert phase in ["train", "eval"], "invalid running phase"
    loss_sum = 0.0
    correct = 0.0

    if phase=="train": model.train()
    elif phase=="eval": model.eval()

    ttl = 0
    with torch.autograd.set_grad_enabled(phase=="train"):
        for i, (input, target) in tqdm(enumerate(loader), total=len(loader)):
            input = input.to(device=device)
            target = target.to(device=device)
            output = model(input)
            loss = criterion(output, target)
            loss_sum += loss.cpu().item() * input.size(0)
            pred = output.data.max(1, keepdim=True)[1]
            correct += pred.eq(target.data.view_as(pred)).sum()
            ttl += input.size()[0]

            if phase=="train":
                loss = loss * 1000 # do loss scaling
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

    correct = correct.cpu().item()
    return {
        'loss': loss_sum / float(ttl),
        'accuracy': correct / float(ttl) * 100.0,
    }
```

Begin the training process just as usual. Enjoy!

[10]:
```
for epoch in range(1):
    train_res = run_epoch(loaders['train'], model, F.cross_entropy,
                          optimizer=optimizer, phase="train")
    test_res = run_epoch(loaders['test'], model, F.cross_entropy,
                          optimizer=optimizer, phase="eval")
```

```
100%|| 391/391 [00:14<00:00, 26.41it/s]
100%|| 79/79 [00:01<00:00, 78.18it/s]
```

[11]:
```
train_res
```

[11]:
```
{'loss': 1.6471979439544677, 'accuracy': 37.566}
```

[12]:
```
test_res
```

[12]:
```
{'loss': 1.5749474658966065, 'accuracy': 43.63}
```

CHAPTER 3

qtorch package

class qtorch.**FixedPoint**(*wl*, *fl*, *clamp=True*, *symmetric=False*)

Low-Precision Fixed Point Format. Defined similarly in *Deep Learning with Limited Numerical Precision* (https://arxiv.org/abs/1502.02551)

The representable range is $[-2^{wl-fl-1}, 2^{wl-fl-1} - 2^{-fl}]$ and a precision unit (smallest nonzero absolute value) is 2^{-fl}. Numbers outside of the representable range can be clamped (if *clamp* is true). We can also give up the smallest representable number to make the range symmetric, $[-2^{wl-fl-1-fl}, 2^{wl-fl-1} - 2^{-fl}]$. (if *symmetric* is true).

Define $\lfloor x \rfloor$ to be the largest representable number (multiples of 2^{-fl}) smaller than x. For numbers within the representable range, fixed point quantizatio corresponds to

$$NearestRound(x) = \begin{cases} \lfloor x \rfloor, \text{ if } \lfloor x \rfloor \leq x \leq \lfloor x \rfloor + 2^{-fl-1} \\ \lfloor x \rfloor + 2^{-fl}, \text{ if } \lfloor x \rfloor + 2^{-fl-1} < x \leq \lfloor x \rfloor + 2^{-fl} \end{cases}$$

or

$$StochasticRound(x) = \begin{cases} \lfloor x \rfloor, \text{ with probabilty } 1 - \frac{x - \lfloor x \rfloor}{2^{-fl}} \\ \lfloor x \rfloor + 2^{-fl}, \text{ with probabilty } \frac{x - \lfloor x \rfloor}{2^{-fl}} \end{cases}$$

Args:

- **attr** wl (int) : word length of each fixed point number
- **attr** fl (int) : fractional length of each fixed point number
- **attr** clamp (bool) : whether to clamp unrepresentable numbers
- **attr** symmetric (bool) : whether to make the representable range symmetric

class qtorch.**BlockFloatingPoint**(*wl*, *dim=-1*)

Low-Precision Block Floating Point Format.

BlockFloatingPoint shares an exponent across a block of numbers. The shared exponent is chosen from the largest magnitude in the block.

Args:

- **attr** *wl* word length of the tensor
- **attr** *dim* block dimension to share exponent. (*, D, *) Tensor where

  D is at position *dim* will have D different exponents; use -1 if the entire tensor is treated as a single block (there is only 1 shared exponent).

class qtorch.**FloatingPoint**(*exp*, *man*)

Low-Precision Floating Point Format.

We set the exponent bias to be 2^{exp-1}. In our simulation, we do not handle denormal/subnormal numbers and infinities/NaNs. For rounding mode, we apply *round to nearest even*.

Args:

- **attr** *exp*: number of bits allocated for exponent
- **attr** *man*: number of bits allocated for mantissa, referring to number of bits that are supposed to be stored on hardware (not counting the virtual bits).

CHAPTER 4

qtorch.quant package

qtorch.quant.**fixed_point_quantize**(*x*, *wl*, *fl*, *clamp=True*, *symmetric=False*, *rounding='stochastic'*)

Quantize a single precision Floating Point into low-precision Fixed Point

Args:

- **param** *x* (torch.Tensor) : the single precision number to be quantized
- **param** *wl* (int) : word length of the fixed point number being simulated
- **param** *fl* (int) : fractional length of the fixed point number being simulated
- **param** *clamp* (bool, optional) : clamp input numbers into representable range. if false, the quantization will only simulate the effect on precision
- **param** *symmetric* (bool, optional) : discard the minimum representable number to make the representable range symmetric
- **param** *rounding* (string) : rounding mode, "stochastic" or "nearest" (default: "stochastic")

Returns:

- a quantized low-precision block floating point number (torch.Tensor)

qtorch.quant.**block_quantize**(*x*, *wl*, *dim=-1*, *rounding='stochastic'*)

Quantize a single precision Floating Point into low-precision Block Floating Point

Args:

- **param** *x* (torch.Tensor) : the single precision number to be quantized
- **param** *wl* (int) : word length of the block floating point number being simulated
- **param** *rounding* (string) : rounding mode, "stochastic" or "nearest"

Returns:

- a quantized low-precision block floating point number (torch.Tensor)

qtorch.quant.**float_quantize**(*x*, *exp*, *man*, *rounding='stochastic'*)

Quantize a single precision Floating Point into low-precision Floating Point

Args:

- **attr** *x* (torch.Tensor) : the single precision number(torch.Tensor) to be quantized
- **attr** *exp* (int) : number of bits allocated for exponent
- **attr** *man* (int) : number of bits allocated for mantissa, not counting the virtual bit
- **attr** *rounding* (string) : rounding mode, "stochastic" or "nearest"

Returns:

- a quantized low-precision floating point number (torch.Tensor)

qtorch.quant.**quantizer**(*forward_number=None*, *backward_number=None*, *forward_rounding='stochastic'*, *backward_rounding='stochastic'*, *clamping_grad_zero=False*, *backward_hooks=[]*)

Creates a quantization function to support quantizing forward and backward process differently.

Args:

- **param** forward_number (qtorch.Number, optional) : the number format used for forward quantization. if is None, the quantization would be a identity mapping.
- **param** backward_number (qtorch.Number, optional) : the number format used for backward quantization. if is None, the quantization would be a identity mapping.
- **param** forward_rounding (string) : rounding mode, "stochastic" or "nearest" (default: "stochastic")
- **param** backward_rounding (string) : rounding mode, "stochastic" or "nearest" (default: "stochastic")
- **param** clamping_grad_zero (bool) : zero out the gradient of numbers that are being clamped during forward propagation. currently requires forward_number to be a fixed point number.
- **param** backward_hooks (iterable) : iterable of functions that will be applied to gradients before backward quantization. For example, this can be used to support custom scaling.

Returns: A quantization function as specified (torch.Tensor -> torch.Tensor)

class qtorch.quant.**Quantizer**(*forward_number=None*, *backward_number=None*, *forward_rounding='stochastic'*, *backward_rounding='stochastic'*)

Bases: ModuleMock

forward(*x*)

CHAPTER 5

qtorch.optim package

CHAPTER 6

qtorch.auto_low

qtorch.auto_low.**lower**(*model*, *layer_types=[]*, *forward_number=None*, *backward_number=None*, *forward_rounding='stochastic'*, *backward_rounding='stochastic'*)

qtorch.auto_low.**sequential_lower**(*model*, *layer_types=[]*, *forward_number=None*, *backward_number=None*, *forward_rounding='stochastic'*, *backward_rounding='stochastic'*)

Return a new model without touching the old one

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

q

Index

B

F

L

Q

S