ELEC5306 Project2 Section 1.5 Bonus Report

Group 4

Abstract

In this section, we set up a new network so that it could implement the quantization process. The method is adding a new layer in the network (after convolution layers) as a binary quantizer.

This implementation method refers to the paper published by Matthieu, named "Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or −1" [1].

1. Background

In this paper, they mentioned two different binarization functions:

$$x^b = Sign(x) = \begin{cases} +1, & x \geq 0 \\ -1, & otherwise \end{cases} \quad (1)$$

where x^b is the binarized variable (weight or activation) and x is the real-valued variable.

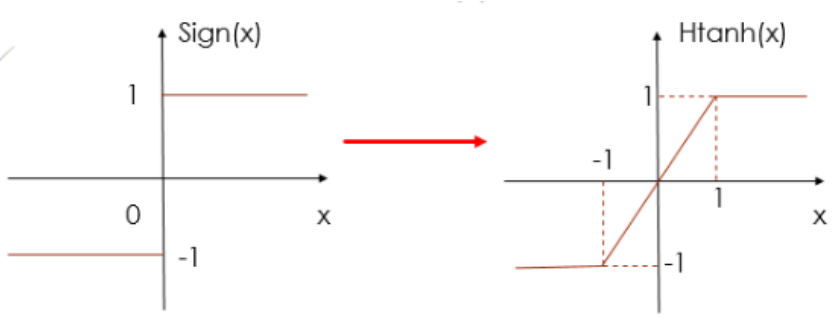
The second binarization function is stochastic

$$x^b = Sign(x) = \begin{cases} +1, & \text{with probability p } = \sigma(x) \\ -1, & \text{with probability 1 } - \text{ p} \end{cases} \quad (2)$$

where σ is the "hard sigmoid" function.

It has been claimed that the stochastic binarization (second function) is more appealing than the sign function (first function). However, random values are hard to implement and rely on the performance of the hardware. In this case, we finally used the first sign function as a quantization function.

Besides, there will be a problem if we used the sign function directly. It is because the gradients of the Sign(x) function are all zero, so the network cannot be trained by gradient descent. So in the article, they used a Hard-tanh function (Figure below) to approximate the Sign(x) function, so that during the backward propagating, the gradient can be calculated.



1 Sign vs Htanh

2. Implement

As our implementation, we firstly define a new layer class so that it could implement the quantization function. There is a similar open-source implementation for the paper on Github done by Ding Ke [2].

However, he pre-processes the data so that it will change the data type from tensor to NumPy, which could be used for `binarize()` function in sklearn.preprocessing directly to assign the value (figure below) [2].

```
10  class BinaryTanh(nn.Module):
11      def __init__(self):
12          super(BinaryTanh, self).__init__()
13          self.hardtanh = nn.Hardtanh()
14
15      def forward(self, input):
16          output = self.hardtanh(input)
17          output = binarize(output)
18          return output
```

2 Open-source code for Binary-Tanh

Besides, the output after binarize() function should be either 0 or 1, which is slightly different with the original paper which is either -1 or 1. But the aim is the same, which could both implement the binary quantization.

However, our inputs are all tensors, so we used Threshold() function in PyTorch [3] to replace binarize() function to help us assign the binary value (Figure below).

$$y = \begin{cases} x, & \text{if } x > \text{threshold} \\ \text{value}, & \text{otherwise} \end{cases}$$

3 PyTorch Threshold function

According to the official manual of sklearn [4], the default threshold is 0. In this case, feature values below or equal to this threshold are replaced by 0, above it by 1.

In this case, we also set the threshold as 0 for Threshold() function, and our new layer for quantization is shown below:

```
from sklearn.preprocessing import binarize
class BinaryTanh(nn.Module):
        def __init__(self):
                super(BinaryTanh, self).__init__()
                self.hardtanh = nn.Hardtanh()

        def forward(self, input):
                output = self.hardtanh(input)
                threshold = torch.tensor([0]).cuda()
                results = (output>threshold).float()*1
                # output = binarize(output)
                return results
```

4 Binary-Tanh function

Based on the figure below, with this self-defined layer (g_q), we could use it to quantize the compressed value after using 4 convolution layers (g_a). Then, make the quantized output as the input for the de-convolution layers (g_s) in the forward process. The detailed network code is shown below:

```
class Network(nn.Module):

        def __init__(self, N, M, init_weights=True, **kwargs):
                super().__init__(**kwargs)

                self.g_a = nn.Sequential(
                        conv(3, N),
                        conv(N, N),
                        conv(N, N),
                        conv(N, M),
                )

                self.g_q = nn.Sequential(
                          BinaryTanh(),

                )

                self.g_s = nn.Sequential(
                        deconv(M, N),
                        deconv(N, N),
                        deconv(N, N),
                        deconv(N, 3),
                )

                self.N = N
                self.M = M

                if init_weights:
                        self._initialize_weights()

        def forward(self, x):
                y = self.g_a(x)
                quan = self.g_q(y)
                x_hat = self.g_s(quan)
                return {
                        "x_hat": x_hat,
                        # "x_quan": quan,
                }
```

5 New network

Besides, in this network, we also defined a new function called “quantize”, similar with

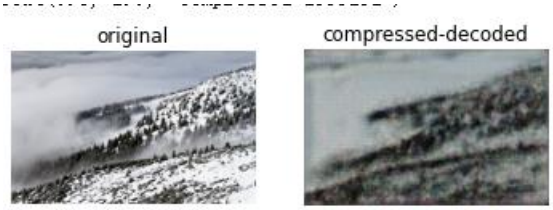
the compressed function, which will return the value after quantization so that we could find the result after quantization.

```
def compress(self, x):
        y = self.g_a(x)
        return y

def decompress(self, y_hat):
        x_hat = self.g_s(y_hat).clamp_(0, 1)
        return {"x_hat": x_hat}

def quantize(self, x):
    y = self.g_a(x)
    quan = self.g_q(y)
    return quan
```

6 Three functions based on the network

3. Results

After training for 10 epochs, we could compare the original image and compressed-decoded image as below:



Besides, we could compare the compressed result and quantization after it as below:

```
print(compressed)
```

```
tensor([[[[-0.7899,  0.3098, -0.1982,  ..., -0.1204, -0.1482, -0.7139],
          [-1.9694, -2.0050, -1.6346,  ..., -1.5357, -1.6562, -1.4790],
          [-2.2714, -3.1026, -2.9277,  ..., -2.7735, -2.6594, -2.3210],
          ...,
          [-1.8174, -3.4641, -3.2613,  ..., -3.7026, -3.6824, -2.7426],
          [-0.8735, -3.1516, -0.4203,  ..., -3.4498, -3.8696, -3.4357],
          [-2.1765,  0.6771, -0.8890,  ..., -3.4441, -2.9102, -2.7229]],

         [[-1.4193,  0.4435, -0.3559,  ..., -0.4265, -0.4473,  1.0944],
          [-0.0464,  2.0199,  1.0392,  ...,  0.9209,  0.9196,  2.4052],
          [-0.7236,  0.1885, -0.3894,  ..., -0.2110, -0.2783,  1.2323],
          ...,
          [ 0.0580,  0.4822, -0.5186,  ..., -0.5947,  0.3536,  1.3152],
          [-0.0150, -0.3185,  0.2680,  ..., -0.5512, -0.7478,  1.6163],
          [-0.9031, -0.3164, -1.2413,  ..., -0.4043, -0.3186,  0.6187]],

         [[-1.6368, -1.6221, -2.3080,  ..., -2.4744, -2.5025, -0.3788],
          [-0.7903,  0.0752, -0.6551,  ..., -0.6507, -0.6625,  0.9980],
          [-0.9403, -0.7016, -1.0990,  ..., -1.0586, -1.1171,  0.3539],
          ...,
          [ 0.2305, -0.6097, -0.3501,  ..., -0.7270, -1.5073,  0.4039],
          [-0.3145, -0.5725, -1.7377,  ..., -1.9348, -1.8557,  0.3576],
          [ 1.3820,  1.8601,  1.0619,  ...,  1.3271,  1.0970,  0.5666]],

         ...,
```

```
print(quantized)
```

```
tensor([[[[0., 1., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 1., 0.,  ..., 0., 0., 0.]],

         [[0., 1., 0.,  ..., 0., 0., 1.],
          [0., 1., 1.,  ..., 1., 1., 1.],
          [0., 1., 0.,  ..., 0., 0., 1.],
          ...,
          [1., 1., 0.,  ..., 0., 1., 1.],
          [0., 0., 1.,  ..., 0., 0., 1.],
          [0., 0., 0.,  ..., 0., 0., 1.]],

         [[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 1., 0.,  ..., 0., 0., 1.],
          [0., 0., 0.,  ..., 0., 0., 1.],
          ...,
          [1., 0., 0.,  ..., 0., 0., 1.],
          [0., 0., 0.,  ..., 0., 0., 1.],
          [1., 1., 1.,  ..., 1., 1., 1.]],

         ...,
```

Also, the compressed ratio which is $Image_{quantized}/Image_{original}$ is calculated by using the template code below:

```
print('quantization ratio: quantized/original {}'.format(quantized.numel() / image.numel()))
```

```
quantization ratio: quantized/original 0.25165562913907286
```

Reference:

[1]: Courbariaux M, Bengio Y. Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1[J]. arXiv preprint arXiv:1602.02830, 2016.

[2]: Ding Ke: An implementation of binaryNet pytorch. Referred from: https://github.com/DingKe/pytorch_workplace/tree/master/binary

[3]: Pytorch THRESHOLD. Referred from: https://pytorch.org/docs/stable/generated/torch.nn.Threshold.html

[4]: sklearn.preprocessing.binarize(). Referred from: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.binarize.html