Deep Learning Project using Resnet and CIFAR-10

Seongmin Kim
KyungHee University
2021105575
briankim0809@khu.ac.kr

Abstract

The original code was optimized for ImageNet, which has larger images (224x224) and 1000 classes. In contrast, CIFAR-10 has smaller images (32x32) and only 10 classes. Thus, the code was modified to ensure efficiency and alignment with ResNet-20 architecture by adjusting the network to use three stages of the appropriate filter size to match the structure proposed in the paper for CIFAR-10. As a result of modifying the code, the accuracy and number of parameters have become quite similar to the paper.

The version I modified has a parameter count of 272474 (0.27M) and the paper is 0.27M. The version I modified had 92.07 percent accuracy and the paper was 91.25percent = 100-8.75.

1. Introduction

This report is based on K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90. [1]

And code is based on kuangliu/pytorch-cifar github (https://github.com/kuangliu/pytorch-cifarhttps://github.com/kuangliu/pytorch-cifar) [2]

1.1. Paper summary

In the existing CNN model, the vanishing gradient problem occurred when the layers were stacked deeply, and the accuracy was rather degraded. This phenomenon is different from overfitting, but unlike overfitting, the degradation problem lowers both training and test accuracy. Instead of hoping each few stacked layers directly fit a desired underlying mapping, we explicitly let these layers fit a residual mapping to solve degradation problem. That is, which skip one or more layers that skip one or more layers. As a result, it can accumulate deeper layers compared to existing.

1.2. Issues

First, we need to know the difference between ImageNet and CIFAR-10 datasets. Both are famous datasets, but Imagenet is a dataset containing a total of over a million data with an average resolution of 469x387 consisting of 1,000 classes. CIFAR-10 is a dataset containing 60,000 data with a 32x32 resolution consisting of 10 classes.

In the code is configured to be suitable for ImageNet, the size of the filter is too large compared to the CIFAR-10 dataset and the layer needs to be reduced. The number of parameters of the model was reduced through adjustments such as reducing the filter size and removing unnecessary steps by referring to the paper. A few additional adjustments were then made to ensure that it was calculated correctly.

The terminology was mixed, and the layer in the code is a different layer name from 'Resnet20 used 20 layers.' Layer in the code means one layer in which several basic blocks are stacked. Or you can call it stage. From now on, I'll call it stage.

2. Code Modification

2.1. Paper Reference for code modification

> The first layer is 3×3 convolutions. Then we use a stack of 6n layers with 3×3 convolutions on the feature maps of sizes 32, 16, 8 respectively, with 2n layers for each feature map size. The numbers of filters are 16, 32, 64 respectively. The subsampling is performed by convolutions with a stride of 2.[1]
>
> When shortcut connections are used, they are connected to the pairs of 3×3 layers (totally 3n shortcuts).[1]
>
> We use a weight decay of 0.0001 and momentum of 0.9[1]

2.2. Code modification - Modify stage

The first place to revise is this part.

> Then we use a stack of 6n layers with 3×3 convolutions on the feature maps of sizes 32, 16, 8 respectively, with 2n layers for each feature map size. When shortcut connections are used, they are connected to the pairs of 3×3 layers (totally 3n shortcuts).[1]

Through this part, I found out that there are three stages, and to create 20 layers, 1 initial convolution layer + 3 stages consisting of 3 Basic Blocks each(each basic block has 2 convolution layers)+final fully connected layer**=1+ 3x3x2+1=20**. So, I wrote the code

```
def ResNet20():
    return ResNet(BasicBlock, [3, 3, 3])
```

2.3. Code modification - Modify filter

Next is the adjustment of the filter.

> The first layer is 3×3 convolutions. Then we use a stack of 6n layers with 3×3 convolutions on the feature maps of sizes 32, 16, 8 respectively, with 2n layers for each feature map size. The numbers of filters are 16, 32, 64 respectively. The subsampling is performed by convolutions with a stride of 2.[1]

Referring to this, I revised it as follows

```
self.in_planes = 16
self.conv1 = nn.Conv2d(3, 16, kernel_size=3,
    stride=1, padding=1, bias=False)
self.bn1 = nn.BatchNorm2d(16)
self.layer1 = self._make_layer(block, 16,
    num_blocks[0], stride=1)
self.layer2 = self._make_layer(block, 32,
    num_blocks[1], stride=2)
self.layer3 = self._make_layer(block, 64,
    num_blocks[2], stride=2)
self.linear = nn.Linear(64 * block.expansion,
    num_classes)
```

and since there are three stages, I deleted this part.

```
self.layer4 = self._make_layer(block, 512,
    num_blocks[3], stride=2)
```

The previous code is

```
self.in_planes = 64
self.conv1 = nn.Conv2d(3, 64, kernel_size=3,
    stride=1, padding=1, bias=False)
self.bn1 = nn.BatchNorm2d(64)
self.layer1 = self._make_layer(block, 64,
    num_blocks[0], stride=1)
self.layer2 = self._make_layer(block, 128,
    num_blocks[1], stride=2)
self.layer3 = self._make_layer(block, 256,
    num_blocks[2], stride=2)
self.layer4 = self._make_layer(block, 512,
    num_blocks[3], stride=2)
self.linear = nn.Linear(512*block.expansion,
    num_classes)
```

2.4. Code modification - Dimension Mismatch

However, this correction causes errors due to **dimensional inconsistencies**. To correct this, the dimensional inconsistency was resolved by adjusting the size of the pooling kernel (in the pooling kernel, a pooling operation is performed to reduce the size of the featuremap by downsampling the feature map).

Since the spatial dimensions of the feature maps after the third residual block were larger than expected (8x8), to resolve dimension mismatch issues, I increased the pooling kernel size from 4 to 8, ensuring the final spatial dimensions are reduced to 1x1 before passing into the fully connected layer.

So, I modified the code

```
out = F.avg_pool2d(out, 8)
```

2.5. Code modification -Weight Decay

> We use a weight decay of 0.0001 and momentum of 0.9[1]

Referring to this, I lowered this figure from 5e-4 to 1e-4

```
optimizer = optim.SGD(net.parameters(), lr=args.
    lr,momentum=0.9, weight_decay=1e-4)
```

3. Number of parameters

3.1. Check parameter count code

```
net = net.to(device)
total_params = sum(p.numel() for p in net.
    parameters() if p.requires_grad)
print(f'Total number of parameters: {total_params
    }')
```

3.2. Result

```
Total number of parameters: 272474
```

4. Conclusion and Recommendations

4.1. Conclusions

As a result of modifying the code by referring to the paper, the number and accuracy of parameters were derived quite similar to those of the paper.

1. **Number of parameter**
   - My Revisied version: 272474 (0.27M)
   - In paper: 0.27M
2. **Accuracy**
   - My Revisied version: 92.07percent
   - In paper: 91.25percent = 100-8.75

4.2. Recommendations

In accuracy, the accuracy of the model I modified was slightly higher than the paper that of , which is presumed to be an error due to a slight difference in settings that are not exactly in the paper, such as differences in data augmentation methods and pooling kernel.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. 1

[2] kuangliu. pytorch-cifar. https://github.com/kuangliu/pytorch-cifar. 1

A. Result screenshots

```
Total number of parameters: 272474
```

Figure 1. Number of parameters

```
Epoch: 199
 [==============================================================>]  Step: 44ms | Tot:  391/391
 [==============================================================>]  Step: 72ms | Tot: 100/100
Saving..
Best accuracy: 92.07
```

Figure 2. Accuracy

B. The entire code

B.1. Resnet.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride
        =1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes,
             kernel_size=3, stride=stride,
            padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes,
            kernel_size=3, stride=1, padding=1,
            bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.
            expansion * planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.
                    expansion * planes,
                    kernel_size=1, stride=stride,
                     bias=False),
                nn.BatchNorm2d(self.expansion *
                    planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out


class ResNet(nn.Module):
    def __init__(self, block, num_blocks,
        num_classes=10):
        super(ResNet, self).__init__()
        self.in_planes = 16

        self.conv1 = nn.Conv2d(3, 16, kernel_size
            =3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.layer1 = self._make_layer(block, 16,
             num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 32,
             num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 64,
             num_blocks[2], stride=2)
        self.linear = nn.Linear(64 * block.
            expansion, num_classes)

    def _make_layer(self, block, planes,
        num_blocks, stride):
        strides = [stride] + [1] * (num_blocks -
            1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes,
                planes, stride))
            self.in_planes = planes * block.
                expansion
        return nn.Sequential(*layers)

    def forward(self, x):
      out = F.relu(self.bn1(self.conv1(x)))
      out = self.layer1(out)
      out = self.layer2(out)
      out = self.layer3(out)
      out = F.avg_pool2d(out, 8)
      out = out.view(out.size(0), -1)
      out = self.linear(out)
      return out


def ResNet20():
    return ResNet(BasicBlock, [3, 3, 3])


# Test the model
def test():
    net = ResNet20()
    y = net(torch.randn(1, 3, 32, 32))
    print(y.size())

test()
```

B.2. main.py

```
'''Train CIFAR10 with PyTorch.'''
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torch.backends.cudnn as cudnn

import torchvision
import torchvision.transforms as transforms

import os
import argparse



parser = argparse.ArgumentParser(description='
    PyTorch CIFAR10 Training')
parser.add_argument('--lr', default=0.1, type=
    float, help='learning rate')
parser.add_argument('--resume', '-r', action='
    store_true',
                    help='resume from checkpoint'
                        )
args = parser.parse_args(args=[])

device = 'cuda' if torch.cuda.is_available() else
     'cpu'
best_acc = 0  # best test accuracy
start_epoch = 0  # start from epoch 0 or last
    checkpoint epoch

# Data
print('==> Preparing data..')
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)),
])

trainset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True,
        transform=transform_train)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=128, shuffle=True,
        num_workers=2)

testset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True,
        transform=transform_test)
testloader = torch.utils.data.DataLoader(
    testset, batch_size=100, shuffle=False,
        num_workers=2)

classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck
               ')

# Model
print('==> Building model..')
# net = VGG('VGG19')
# net = ResNet18()
# net = PreActResNet18()
# net = GoogLeNet()
# net = DenseNet121()
# net = ResNeXt29_2x64d()
# net = MobileNet()
# net = MobileNetV2()
# net = DPN92()
# net = ShuffleNetG2()
# net = SENet18()
# net = ShuffleNetV2(1)
# net = EfficientNetB0()
# net = RegNetX_200MF()
#net = SimpleDLA()
net=ResNet20()
net = net.to(device)
total_params = sum(p.numel() for p in net.
    parameters() if p.requires_grad)
print(f'Total number of parameters: {total_params
    }')
if device == 'cuda':
    net = torch.nn.DataParallel(net)
    cudnn.benchmark = True

if args.resume:
    # Load checkpoint.
    print('==> Resuming from checkpoint..')
    assert os.path.isdir('checkpoint'), 'Error:
        no checkpoint directory found!'
    checkpoint = torch.load('./checkpoint/ckpt.
        pth')
    net.load_state_dict(checkpoint['net'])
    best_acc = checkpoint['acc']
    start_epoch = checkpoint['epoch']

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=args.
    lr,
                      momentum=0.9, weight_decay
                          =1e-4)
scheduler = torch.optim.lr_scheduler.
    CosineAnnealingLR(optimizer, T_max=200)


# Training
def train(epoch):
    print('\nEpoch: %d' % epoch)
    net.train()
    train_loss = 0
    correct = 0
    total = 0
    for batch_idx, (inputs, targets) in enumerate
        (trainloader):
        inputs, targets = inputs.to(device),
            targets.to(device)
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().
            item()
```

```
        progress_bar(batch_idx, len(trainloader),
             'Loss: %.3f | Acc: %.3f%% (%d/%d)'
                    % (train_loss/(batch_idx+1),
                         100.*correct/total,
                        correct, total))


def test(epoch):
    global best_acc
    net.eval()
    test_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (inputs, targets) in
            enumerate(testloader):
            inputs, targets = inputs.to(device),
                targets.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, targets)

            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum
                ().item()

            progress_bar(batch_idx, len(
                testloader), 'Loss: %.3f | Acc:
                %.3f%% (%d/%d)'
                         % (test_loss/(batch_idx
                             +1), 100.*correct/
                             total, correct,
                             total))

    # Save checkpoint.
    acc = 100.*correct/total
    if acc > best_acc:
        print('Saving..')
        state = {
            'net': net.state_dict(),
            'acc': acc,
            'epoch': epoch,
        }
        if not os.path.isdir('checkpoint'):
            os.mkdir('checkpoint')
        torch.save(state, './checkpoint/ckpt.pth'
            )
        best_acc = acc


for epoch in range(start_epoch, start_epoch+200):
    train(epoch)
    scheduler.step()
test(0)
print('Best accuracy:', best_acc)
```