HW5

- hackmd

Implementation details

Environment

- Using Google colab GPU print(torch.__version__) = 1.11.0+cu113

Main function

working flow

- I run the following main block 3 times, that is, the 1st step, 2nd step, and 3rd step

Main block

```
# create batch data
trainloader = augmentation_on_training_data(x_train)
testloader = normalize_testing_data(x_test)

# initialize model
net = init_model()
criterion, optimizer, scheduler = update_opt(net, mylr)

# start training
for epoch in range(start_epoch, 140):
    train(epoch)
    scheduler.step()

# testing
test()
```

Data preprocess

augmentation & normalization on training data

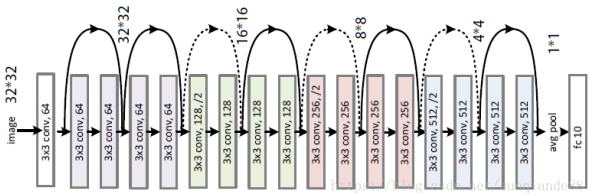
```
# ************************************************** #
# transformation :
# np --> tensor --> PIL --> random crop/flip
#                       --> normalized tensor
# ************************************************** #

class ciphar10_calss:
    def __init__(self, data):
        self.data = data

    def __getitem__(self, index: int):
        return (self.data[index])

    def __len__(self):
        return len(self.data)


def augmentation_on_training_data(x_train):
    transform_train = transforms.Compose([
        transforms.ToTensor(),
        transforms.ToPILImage(),
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
    ])

    _x_train_trans = []
    for i, img in enumerate(x_train):
        new_img = transform_train(img)
        _x_train_trans.append((new_img, train_labels[i]))


    trainloader = torch.utils.data.DataLoader(
        ciphar10_calss(_x_train_trans),
        batch_size=128,
        shuffle=True,
        num_workers=2)

    return trainloader
```

normalization on testing data

```
def normalize_testing_data(x_test, without_label=False):
    transform_test = transforms.Compose([
      transforms.ToTensor(),
      transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
    ])
    ...
```

Model architecture

ResNet18



Reference and problems

References

- [1] and [2] are almost the same
- The reason I choose this model is that they can reach 95% accuracy (but I got only 91% QQ)

Problems

- The original codes use test set during the evaluation
  - But someone reported that he reached 95% without using test set as validation
- In my opinion, the model is a mixure of [3], and [4]

Ref	[3]	[4]
src	**4.2. CIFAR-10 and Analysis** We conducted more studies on the CIFAR-10 dataset [20], which consists of 50k training images and 10k testing images in 10 classes. We present experiments trained on the training set and evaluated on the test set. Our focus is on the behaviors of extremely deep networks, but not on pushing the state-of-the-art results, so we intentionally use simple architectures as follows. The plain/residual architectures follow the form in Fig. 3 (middle/right). The network inputs are 32×32 images, with the per-pixel mean subtracted. 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. The network ends with a global average pooling, a 10-way fully-connected layer, and softmax. There are totally $6n$+2 stacked weighted layers. The following table summarizes the architecture: output map size: 32×32 | 16×16 | 8×8 # layers: 1+2n | 2n | 2n # filters: 16 | 32 | 64 When shortcut connections are used, they are connected to the pairs of 3×3 layers (totally $3n$ shortcuts). On this	layer name | output size | 18-layer conv1 | 112×112 | conv2_x | 56×56 | [3×3, 64; 3×3, 64] ×2 conv3_x | 28×28 | [3×3, 128; 3×3, 128] ×2 conv4_x | 14×14 | [3×3, 256; 3×3, 256] ×2 conv5_x | 7×7 | [3×3, 512; 3×3, 512] ×2 | 1×1 | FLOPs | | 1.8×10^9
	The first layer is 3x3 cov instead of 7x7 in [4]	The numbers of filters are {64, 128, 256 512}, instead of {16, 32, 64} in [3]

Code

```
def ResNet18():
    return ResNet(BasicBlock, [2, 2, 2, 2])


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))


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

Hyperparameters

- Data augmentation and shuffle may lead to different training data.
- Therefore, I ==preprocess the training data 3 times, and then run train() 3 times with different hyperparameters==:
- Note that epoch start from a multiple of 5 since I save checkpoints every 5 epochs.

1st step:

- epoch = start from 0 to 140
- learning rate = 0.1
- Result : Testing accuracy from 0% to 89%

2nd step

- epoch = start from 135 to 148 (detect convergence and stop on 148)
- learning rate = 0.01
- Result : Testing accuracy from 89% to 91%

3rd step

- epoch = start from 145 to 164
- learning rate = 0.001
- Result : Testing accuracy from 91% to 92%

same parameters among 3 steps

- batch_size of training set = 128
- criterion = nn.CrossEntropyLoss()
- optimizer = optim.SGD(net.parameters(), lr=mylr, momentum=0.9, weight_decay=5e-4)

Framwork

pytorch

Accuracy

92%

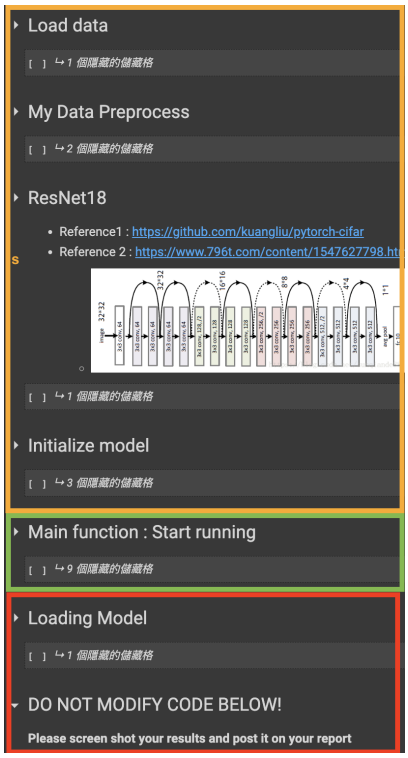
Reproduce the result:

- Put 0616086_HW5.ipynb, utils.py, and ResNet/ under the same directory
  - google drive

Reproduce steps

PLEASE DON'T USE GPU if you just want to reproduce the result.

- Run all yellow blocks
  - these blocks load data and define functions
- Skip the green block
  - these blocks are for training
- Run the Red blocks
  - these block will resume model from ResNet/0001/ckpt_160.pth and do testing



TAGS: PATTERN-RECOGNITION