RCNN

May 19, 2019

0.1 Imports

```
In [36]: import os
         import numpy as np
         import matplotlib.pyplot as plt
         %matplotlib inline

         import torch
         import torchvision
         from torchvision.datasets import ImageFolder
         import torchvision.transforms as transforms

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

```
In [37]: #Setup Torch CUDA torch device
         device = torch.device('cuda:0')
```

```
In [38]: transform = transforms.Compose([
             transforms.ToTensor(),  # Transform to tensor
             transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])
             #transforms.Normalize((0.5,), (0.5,))  # Min-max scaling to [-1, 1]
         ])

         data_dir = os.path.join('fruits')
         print('Data stored in %s' % data_dir)

         trainset = ImageFolder("./fruits/Training",transform=transform)
         testset = ImageFolder("./fruits/Test",transform=transform)
```

```
Data stored in fruits
```

```
In [39]: def generate_labels():
             trainset_labels = []
             testset_labels = []
             for i in trainset.imgs:
                 trainset_labels.append(i[1])
```

```
            for j in testset.imgs:
                testset_labels.append(j[1])

            return (trainset_labels, testset_labels)
```

In [40]:
```
# Total classes
classes_idx_dict = trainset.class_to_idx # {'Class Name': idx }
classes = len(trainset.classes)
len_trainset = len(trainset)
len_testset = len(testset)

train_labels, test_labels = generate_labels()
print(f'Trainset has total of {classes} classes')
```

```
Trainset has total of 103 classes
```

In [41]:
```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=150, shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=150, shuffle=False)

image_shape = iter(trainloader).next()[0].shape
BATCH_SIZE, CHANNELS, HEIGHT, WIDTH = iter(trainloader).next()[0].shape
print(f'Image: batch size={image_shape[0]}, channels={image_shape[1]}, image height={im
```

```
Image: batch size=150, channels=3, image height=100, image width=100
```

In [51]:
```
class RCL(nn.Module):
    def __init__(self, K=192):
        super(RCL, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(K, K, kernel_size=(3, 3), stride=(1,1)),
            nn.ReLU(),
            nn.BatchNorm2d(K),
        )
        self.conv2 = nn.Conv2d(K, K, kernel_size=(3, 3), stride=(1,1),padding=(1,1))
        self.relu = nn.ReLU()
        self.bnorm = nn.BatchNorm2d(K)

    def forward(self, X, verbose=False):
        #T = 0
        conv1 = self.conv1(X)

        #T = 1
        conv2 = self.relu(F.conv2d(input=conv1,weight=self.conv2.weight, padding=(1,1))
        rcl2 = torch.add(conv1, conv2)
        bn2 = self.bnorm(rcl2)
```

```
        #T = 2
        conv3 = self.relu(F.conv2d(input=bn2,weight=self.conv2.weight, padding=(1,1)))
        rcl3 = torch.add(conv1, conv3)
        bn3 = self.bnorm(rcl3)

        #T = 3
        conv4 = self.relu(F.conv2d(input=bn3,weight=self.conv2.weight, padding=(1,1)))
        rcl4 = torch.add(conv1, conv4)
        bn4 = self.bnorm(rcl4)

        return bn4

class RCNNet(nn.Module):
    def __init__(self, cls, K=192):
        super(RCNNet, self).__init__()
        '''
        To save computation, layer 1 is the standard feed-forward convolutional layer
        without recurrent connections, followed by max pooling.
        Layers 1 to 5 were constrained to have the same number of feature maps K.
        Kernel size in layer 1 is 5 × 5, the feed-forward and recurrent filter sizes i
        Dropout is used after each RCL except layer 5, which was connected to the softm
        '''
        cnn = nn.Sequential()
        cnn.add_module(f'Layer 1 (convolution)', nn.Conv2d(3, K, kernel_size=(5,5)))

        # Both pooling operations have stride 2 and size 3
        cnn.add_module(f'max pooling', nn.MaxPool2d(kernel_size=3, stride=2))

        #  On top of this, four RCLs are used
        cnn.add_module(f'Layer 2 (recurrent convolution)', RCL(K))
        cnn.add_module('Dropout', nn.Dropout(p=0.8))
        cnn.add_module(f'Layer 3 (recurrent convolution)', RCL(K))

        # with a max pooling layer in the middle
        cnn.add_module(f'max pooling', nn.MaxPool2d(kernel_size=3, stride=2))

        # If the RCL was followed by a pooling layer, dropout was placed after pooling.
        cnn.add_module('Dropout', nn.Dropout(p=0.8))

        cnn.add_module(f'Layer 4 (recurrent convolution)', RCL(K))
        cnn.add_module('Dropout', nn.Dropout(p=0.8))
        cnn.add_module(f'Layer 5 (recurrent convolution)', RCL(K))

        self.logsoftmax = nn.LogSoftmax()
        self.cnn = cnn
        self.out = nn.Linear(K, cls)

    def forward(self, input, verbose=False):
```

```
            output = self.cnn(input)
            output = F.max_pool2d(output, kernel_size=(output.shape[1],output.shape[2]),str
            output = output.view(output.size(0), -1)
            output = self.logsoftmax(output)
            output = self.out(output)
            return output
```

In [52]:
```
# Let's test the shapes of the tensors
net = RCNNet(classes,K=32)
net.to(device)
print(net)
with torch.no_grad():
    dataiter = iter(trainloader)
    images, labels = dataiter.next()
    images = images.to(device)
    print('Shape of the input tensor:', images.shape)

    y = net(images, verbose=True)
    print(y.shape)
    assert y.shape == torch.Size([BATCH_SIZE, classes]), f'Bad shape of y: y.shape={y.s

print('The shapes seem to be ok.')
```

```
RCNNet(
  (logsoftmax): LogSoftmax()
  (cnn): Sequential(
    (Layer 1 (convolution)): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1))
    (max pooling): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (Layer 2 (recurrent convolution)): RCL(
      (conv1): Sequential(
        (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
        (1): ReLU()
        (2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (relu): ReLU()
      (bnorm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (Dropout): Dropout(p=0.8)
    (Layer 3 (recurrent convolution)): RCL(
      (conv1): Sequential(
        (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
        (1): ReLU()
        (2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (relu): ReLU()
      (bnorm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
    )
    (Layer 4 (recurrent convolution)): RCL(
      (conv1): Sequential(
        (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
        (1): ReLU()
        (2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (relu): ReLU()
      (bnorm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (Layer 5 (recurrent convolution)): RCL(
      (conv1): Sequential(
        (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
        (1): ReLU()
        (2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (relu): ReLU()
      (bnorm): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (out): Linear(in_features=32, out_features=103, bias=True)
)
Shape of the input tensor: torch.Size([150, 3, 100, 100])
32 39
torch.Size([150, 103])
The shapes seem to be ok.
```

```
C:\Users\User\AppData\Local\conda\conda\envs\dle\lib\site-packages\ipykernel_launcher.py:78: Use
```

In [53]:
```python
def compute_accuracy(net, testloader):
    net.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in testloader:
            images, labels = images.to(device), labels.to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return correct / total
```

In [54]:
```python
initial_learning_rate = 0.001
final_learning_rate = 0.00001
```

```
learning_rate = initial_learning_rate

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=learning_rate)
```

In [56]:
```
n_epochs=10
net.train()
for epoch in range(n_epochs):
    running_loss = 0.0
    print_every = 100  # mini-batches
    for i, (inputs, labels) in enumerate(trainloader, 0):
        # Transfer to GPU
        inputs, labels = inputs.to(device), labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if (i % print_every) == (print_every-1):
            print('[%d, %5d] loss: %.3f' % (epoch+1, i+1, running_loss/print_every))
            running_loss = 0.0

    # Print accuracy after every epoch
    accuracy = compute_accuracy(net, testloader)
    print(f'Accuracy of the network on the {len_testset} test images: {100 * accuracy:.

print('Finished Training')
```

```
C:\Users\User\AppData\Local\conda\conda\envs\dle\lib\site-packages\ipykernel_launcher.py:78: Use


[1,   100] loss: 0.727
[1,   200] loss: 0.315
[1,   300] loss: 0.233
Accuracy of the network on the 17845 test images: 60.611%
[2,   100] loss: 0.124
[2,   200] loss: 0.019
[2,   300] loss: 0.027
Accuracy of the network on the 17845 test images: 97.405%
[3,   100] loss: 0.027
[3,   200] loss: 0.010
```

```
[3,   300] loss: 0.012
Accuracy of the network on the 17845 test images: 96.425%
[4,   100] loss: 0.055
[4,   200] loss: 0.082
[4,   300] loss: 0.043
Accuracy of the network on the 17845 test images: 96.542%
[5,   100] loss: 0.026
[5,   200] loss: 0.006
[5,   300] loss: 0.003
Accuracy of the network on the 17845 test images: 99.159%
[6,   100] loss: 0.000
[6,   200] loss: 0.000
[6,   300] loss: 0.000
Accuracy of the network on the 17845 test images: 99.159%
[7,   100] loss: 0.000
[7,   200] loss: 0.000
[7,   300] loss: 0.000
Accuracy of the network on the 17845 test images: 99.137%
[8,   100] loss: 0.000
[8,   200] loss: 0.000
[8,   300] loss: 0.000
Accuracy of the network on the 17845 test images: 99.115%
[9,   100] loss: 0.000
[9,   200] loss: 0.000
[9,   300] loss: 0.000
Accuracy of the network on the 17845 test images: 99.109%
[10,   100] loss: 0.000
[10,   200] loss: 0.000
[10,   300] loss: 0.000
Accuracy of the network on the 17845 test images: 99.120%
Finished Training
```

In [61]:
```
accuracy = compute_accuracy(net, testloader)
print(f'Accuracy of the network on the test images: {accuracy:.3f}')
```

```
C:\Users\User\AppData\Local\conda\conda\envs\dle\lib\site-packages\ipykernel_launcher.py:78: Use


Accuracy of the network on the test images: 0.991
```

In [57]:
```
filename = 'rcnn.pth'

try:
    do_save = input('Do you want to save the model (type yes to confirm)? ').lower()
    if do_save == 'yes':
        torch.save(net.state_dict(), filename)
        print('Model saved to %s' % filename)
```

```
            else:
                print('Model not saved')
        except:
            raise Exception('The notebook should be run or validated with skip_training=True.')
```

```
Do you want to save the model (type yes to confirm)? yes
Model saved to rcnn.pth
```

```
In [ ]: net = RCNNet()
        net.load_state_dict(torch.load(filename, map_location=lambda storage, loc: storage))
        net.to(device)
        print(f'Model loaded from {filename}')
```

```
In [ ]:
```