Module 32: CNNs in Pytorch

```
import torch
import matplotlib.pyplot as plt
import numpy as np.
```

① Data loading → Loading the CIFAR10 dataset using torchvision

```
import torchvision
import torchvision.transforms as transforms
```

↗ store address

```
trainset = torchvision.datasets.CIFAR10 (root = './data',
                                         train = True,
                                         download = True,
                                         transform = transforms.ToTensor())
```

transforms : applying fns at the time of loading itself

↓

Here, we are converting them to tensor type

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

```
trainloader = torch.utils.data.DataLoader (trainset,
                                           batch_size = 4,
                                           shuffle = True)
```

```
dataiter = iter (trainloader)
```

→ way of working with trainloader is to make an iterator on it

```
images, labels = dataiter.next()
print(images.shape)  # [4, 3, 32, 32]
```

(2) visualize data : To plot image, we convert them into numpy.

```
img = images[0]
npimg = img.numpy()
npimg = npimg.transpose(npimg, (1,2,0))
```

this is the order required to plot the image.

for high order matrix, transpose means any permutation of the different axes

```
plt.figure(figsize = (1,1))
plt.imshow(npimg)
plt.show()
```

move this to a function

```
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1,2,0)))
    plt.show()

imshow(torchvision.utils.make_grid(images))
print(' '.join(classes[labels[j]] for j in range(4)))
```

converts into a repr, where the images get a grid repr.

③ CNNs in Pytorch

1. single convolutional layer

```
class firstCNN (nn.Module):

    def __init__ (self):
        super (firstCNN, self).__init__()
        self.conv1 = nn.Conv2d (3, 16, 3)

    def forward (self, x):
        return self.conv1 (x)

net = firstCNN()
out = net (images)
out.shape → ([4, 16, 30, 30])
```

i/p channels

o/p channels

kernel size

tuple for rest.

to include padding, stride

padding = (1, 1)

stride = (2, 2) → Y, X

*) accessing parameters of model.

```
for param in net.parameters():
    print (param.shape)
```

torch.size ([16, 3, 3, 3])

- 3 → The parameters associated with one kernel along i/p depth
- 16 → The no. of output kernels.

torch.size ([16])

→ The bias

*) plotting representation corresponding to first kernel in the convolution.

```
out1 = out[0, 0, :, :].detach().numpy()
print(out1.shape)
```

2. Deep Convolutional Network

```
class firstCNN_v2(nn.Module):
    def __init__(self):
        super(firstCNN_v2, self).__init__()
        self.model = nn.Sequential(
                        nn.Conv2D(3, 8, 3),
                        nn.Conv2D(8, 16, 3)
                    )
# (N, 3, 32, 32) → (N, 8, 30, 30)
# (N, 8, 30, 30) → (N, 16, 28, 28)
    def forward(self, x):
        return self.model(x)

net = firstCNN_v2()
out = net(image)
out.shape

plt.imshow(out[0, 0, :, :].detach().numpy())
```

$\frac{32-5}{} + 1$

```
class firstCNN_v3(nn.Module):
    def __init__(self):
        super(firstCNN_v3, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3, 6, 5),
            nn.AvgPool2d(2, stride=2),
            nn.Conv2d(6, 16, 5),
            nn.AvgPool2d(2, stride=2)
        )
```

mask of 2*2 → avoids any overlap.

```
# (N, 3, 32, 32) → (N, 6, 28, 28)
# (N, 6, 28, 28) → (N, 6, 14, 14)
# (N, 6, 14, 14) → (N, 16, 10, 10)
# (N, 6, 10, 10) → (N, 6, 5, 5)
```

(4) LeNet

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.cnn_model = nn.Sequential(
            nn.Conv2d(3, 6, 5)
            nn.Tanh()
            nn.AvgPool(2, stride=2)
            nn.Conv2d(6, 16, 5)
            nn.Tanh()
            nn.AvgPool(2, stride=2)
        )
```

```
self.fc_model = nn.Sequential(
        nn.Linear(400, 120),
        nn.Tanh()
        nn.Linear(120, 84),
        nn.Tanh()
        nn.Linear(84, 10)
    )
```

good practice to mention shape modifications.

```
def forward(self, x):
    print(x.shape)
    x = self.cnn_model(x)
    print(x.shape)
    x = x.view(x.size(0), -1)
    print(x.shape)
    x = self.fc_model(x)
    print(x.shape)
    return x
```

batch dim kept same

→ flattening operation

→ softmax not used, already taken into consideration by cross entropy loss for training

```
max_values, pred_class = torch.max(out.data, 1)
print(pred_class)
```

(5) Training Lenet

```
batch_size = 128
trainset = torchvision.datasets.CIFAR10(root = './data',
                                        train = True,
                                        download = True,
```

```
                                transform = transforms.ToTensor())

trainloader = torch.utils.data.DataLoader(trainset,
                                          batch_size=batch_size,
                                          shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data',
                                       train=False,
                                       download=True,
                                       transform=transforms.ToTensor())

testloader = torch.utils.data.DataLoader(testset,
                                         batch_size=batch_size,
                                         shuffle=False)

def evaluation(dataloader):
    total, correct = 0, 0
    for data in dataloader:
        inputs, labels = data
        outputs = net(inputs)
        _, pred = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (pred==labels).sum().item()
    return correct / total * 100

net = LeNet()
```

```
import torch.optim as optim

loss_fn = nn.CrossEntropyLoss()
opt = optim.Adam(net.parameters())
```

*) learning loop :

```
%%time
loss_arr = []
loss_epoch_arr = []
max_epochs = 16

for epoch in range(max_epochs):
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        opt.zero_grad()
        outputs = net(inputs)
        loss = loss_fn(outputs, labels)
        loss.backward()
        opt.step()

        loss_arr.append(loss.item())

    loss_epoch_arr.append(loss.item())

    print('text' %(epoch, max_epochs, evaluation(
                testloader), evaluation(
                trainloader))
```

```
plt.plot(loss_epoch_arr)
plt.show()
```

⑥ Move to GPU → (net, i/p, o/p on GPU)

```
device = torch.device("cuda:0" if torch.cuda.is_available()
                      else "cpu")

def evaluation(dataloader):
    total, correct = 0, 0
    for data in dataloader:
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = net(inputs)
        _, pred = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (pred == labels).sum().item()
    return correct / total * 100

net = LeNet().to(device)
loss_fn = nn.CrossEntropyLoss()
opt = optim.Adam(net.parameters())
%%time
max_epochs = 16
for epoch in range(max_epochs):
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
```

```
        inputs, labels = inputs.to(device), labels.to(device)

        opt.zero_grad()

        outputs = net(inputs)
        loss = loss_fn(outputs, labels)
        loss.backward()
        opt.step()

        loss_am.append(loss.item())

    loss_epoch_am.append(loss.item())

print('text'%(epoch, max_epochs, evaluation(
                testloader), evaluation(
                trainloader)))
```

⑦ Basic visualisation

```
imshow(torchvision.utils.make_grid(images))

net = net.to("cpu")
out = net(images)          → gives the final output
out = net.cnn_model[0](images)   → sending images through that layer
                ↳ sequence of layers
                getting the first layer
out.shape

image_id = 3
plt.figure(figsize = (2,2))
imshow(images[image_id,])
```

```
plt.figure(figsize=(6,6))
plt.subplot(321)
for i in range(6):
    ax1 = plt.subplot(3,2,i+1)
    plt.imshow(out[image_id,i,:,:].detach().
                numpy(), cmap="binary")
plt.show()
```

(8) Exercises

i) Tanh to ReLU in LeNet

ii) Overfitting observed → use weight_decay in torch.optim to add L2 regularisation.

iii) move LeNet to MNIST from CIFAR10.