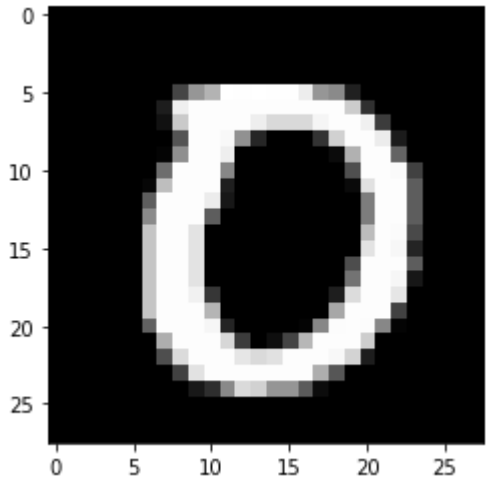
2019-2 인공지능 과제 #2 : 20142697 권민수

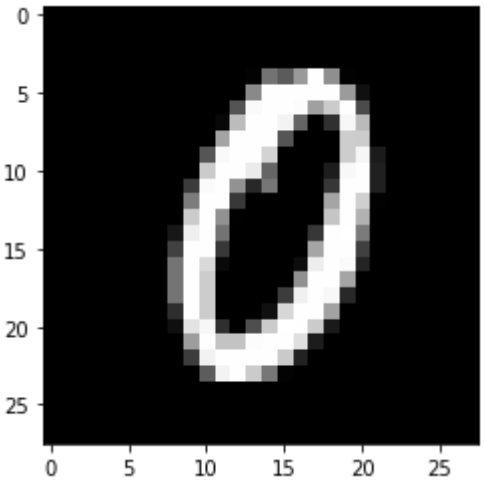
1. 영상인식

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable
import matplotlib.pyplot as plt
%matplotlib inline

is_cuda = False
if torch.cuda.is_available():
    is_cuda = True

transformation =
transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.1307,),
(0.3081,))])

train_dataset = datasets.MNIST('data/', train=True, transform=transformation,
download=True)
test_dataset = datasets.MNIST('data/', train=False, transform=transformation,
download=True)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32,
shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32,
shuffle=True)

sample_data = next(iter(train_loader))

def plot_img(image):
    image = image.numpy()[0]
    mean = 0.1307
    std = 0.3081
    image = ((mean * image) + std)
    plt.imshow(image, cmap='gray')
```

```
plot_img(sample_data[0][2])
# (1) 화면 출력 확인
```




```
plot_img(sample_data[0][1])
# (2) 화면 출력 확인
```




```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        # x = F.dropout(x, p = 0.1, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)

model = Net()
if is_cuda:
    model.cuda()

optimizer = optim.SGD(model.parameters(), lr=0.01)
```

```
data, target = next(iter(train_loader))

output = model(Variable(data.cuda()))
```

```
print(output.size())
# (3) output.size() 출력 확인
```

```
torch.Size([32, 10])
```

```
print(target.size())
# (4) target.size() 출력 확인
```

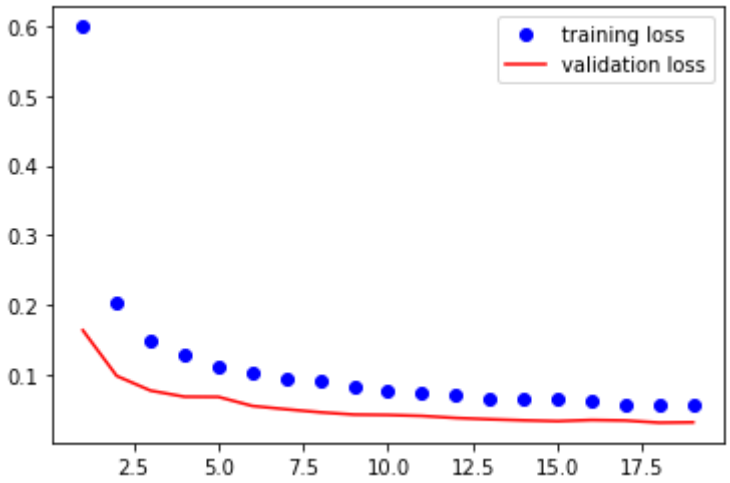
```
 torch.Size([32])
```

```
def fit(epoch, model, data_loader, phase='training', volatile=False):
    if phase == 'training':
        model.train()
    if phase == 'validation':
        model.eval()
        volatile = True
    running_loss = 0.0
    running_correct = 0

    for batch_idx, (data, target) in enumerate(data_loader):
        if is_cuda:
            data, target = data.cuda(), target.cuda()
        data, target = Variable(data, volatile), Variable(target)

        if phase == 'training':
            optimizer.zero_grad()

        output = model(data)
        loss = F.nll_loss(output, target)

        running_loss += F.nll_loss(output, target, size_average=False).item()
        preds = output.data.max(dim=1, keepdim=True)[1]

        running_correct += preds.eq(target.data.view_as(preds)).cpu().sum()
        if phase == 'training':
            loss.backward()
            optimizer.step()

    loss = running_loss/len(data_loader.dataset)
    accuracy = 100. * running_correct/len(data_loader.dataset)

    print(f'{phase} loss is {loss:{5}.{2}} and {phase} accuracy is
{running_correct}/{len(data_loader.dataset)}{accuracy:{10}.{4}}')
    return loss, accuracy

train_losses, train_accuracy = [],[]
val_losses, val_accuracy = [], []

for epoch in range(1,20):
```

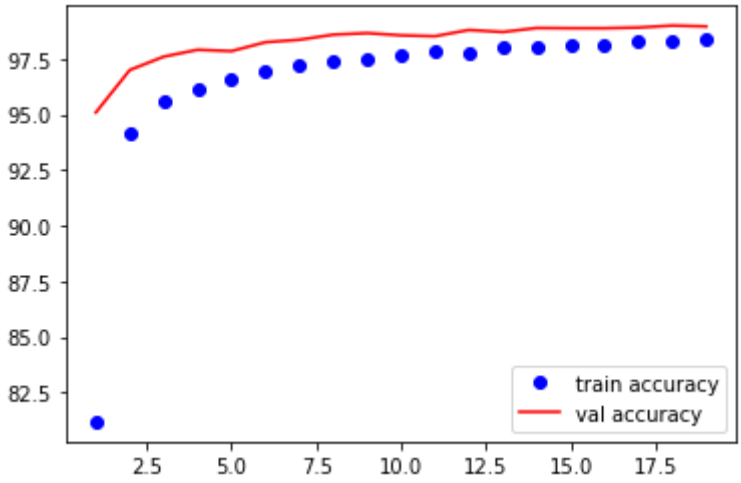
```
    epoch_loss, epoch_accuracy = fit(epoch, model, train_loader,
phase='training')
    val_epoch_loss, val_epoch_accuracy = fit(epoch, model, test_loader,
phase='validation')
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    val_losses.append(val_epoch_loss)
    val_accuracy.append(val_epoch_accuracy)
    # (5) 화면 출력 확인
```

```
training loss is   0.6 and training accuracy is 48706/60000     81.18
validation loss is  0.16 and validation accuracy is 9509/10000     95.09
training loss is   0.2 and training accuracy is 56481/60000     94.14
validation loss is 0.098 and validation accuracy is 9700/10000      97.0
training loss is  0.15 and training accuracy is 57356/60000     95.59
validation loss is 0.077 and validation accuracy is 9760/10000      97.6
training loss is  0.13 and training accuracy is 57681/60000     96.14
validation loss is 0.068 and validation accuracy is 9791/10000     97.91
training loss is  0.11 and training accuracy is 57969/60000     96.61
validation loss is 0.068 and validation accuracy is 9785/10000     97.85
training loss is   0.1 and training accuracy is 58177/60000     96.96
validation loss is 0.055 and validation accuracy is 9825/10000     98.25
training loss is 0.093 and training accuracy is 58333/60000     97.22
validation loss is  0.05 and validation accuracy is 9836/10000     98.36
training loss is  0.09 and training accuracy is 58426/60000     97.38
validation loss is 0.046 and validation accuracy is 9859/10000     98.59
training loss is 0.083 and training accuracy is 58502/60000      97.5
validation loss is 0.042 and validation accuracy is 9866/10000     98.66
training loss is 0.076 and training accuracy is 58609/60000     97.68
validation loss is 0.042 and validation accuracy is 9856/10000     98.56
training loss is 0.074 and training accuracy is 58693/60000     97.82
validation loss is  0.04 and validation accuracy is 9852/10000     98.52
training loss is 0.072 and training accuracy is 58680/60000      97.8
validation loss is 0.037 and validation accuracy is 9880/10000      98.8
training loss is 0.065 and training accuracy is 58827/60000     98.04
validation loss is 0.036 and validation accuracy is 9871/10000     98.71
training loss is 0.064 and training accuracy is 58839/60000     98.07
validation loss is 0.034 and validation accuracy is 9889/10000     98.89
training loss is 0.065 and training accuracy is 58856/60000     98.09
validation loss is 0.033 and validation accuracy is 9888/10000     98.88
training loss is 0.061 and training accuracy is 58870/60000     98.12
validation loss is 0.034 and validation accuracy is 9888/10000     98.88
training loss is 0.058 and training accuracy is 58970/60000     98.28
validation loss is 0.034 and validation accuracy is 9891/10000     98.91
training loss is 0.057 and training accuracy is 58961/60000     98.27
validation loss is 0.031 and validation accuracy is 9900/10000      99.0
training loss is 0.056 and training accuracy is 59009/60000     98.35
validation loss is 0.031 and validation accuracy is 9896/10000     98.96
```

```
plt.plot(range(1, len(train_losses)+1), train_losses, 'bo', label='training
loss')
plt.plot(range(1, len(val_losses)+1), val_losses, 'r', label='validation loss')
plt.legend()
# (6) 화면 출력 확인
```




```
plt.plot(range(1, len(train_accuracy)+1), train_accuracy, 'bo', label='train accuracy')
plt.plot(range(1, len(val_accuracy)+1), val_accuracy, 'r', label='val accuracy')
plt.legend()
# (7) 화면 출력 확인
```



2. CNN

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

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        #  input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation : y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120) # 커널마다 16 * 5 * 5 개의 가중치
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2,2)) # 14 * 14 * 6
```

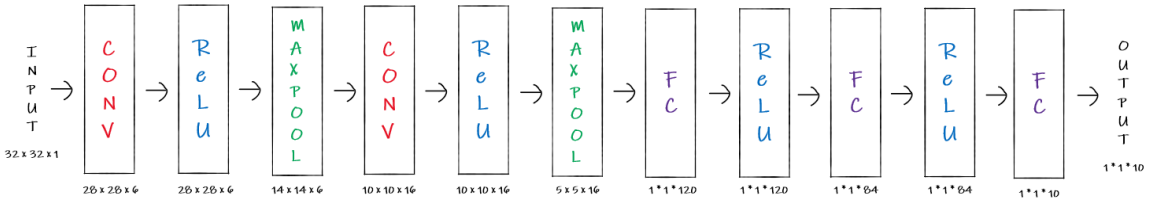
```
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2) # 5 * 5 * 16
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x)) # 1 * 1 * 120
        x = F.relu(self.fc2(x)) # 1 * 1 * 84
        x = self.fc3(x) # 1* 1* 10
        return x

    def num_flat_features(self, x):
        size = x.size()[1:] # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

net = Net()
print(net)
# (1) 화면 출력 확인 및 의미를 서술
```

```
Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

생성한 신경망이 2개의 CONV층, 3개의 FC층을 가지고 있음을 보여주고 있다

- conv1 : input = 1, output = 6, kernel = 5, stride = 1
- conv2 : input = 6, output = 16, kernel = 5, stride = 1
- fc1 : input = 400, output = 84
- fc2 : input = 120, output = 84
- fc3: input = 84, output = 10

```
# (2) 정의된 컨볼루션 신경망의 구조 설명 (위의 AlexNet 그림 참고)
```




```
# net.parameters()를 사용하여 정의된 신경망의 학습가능한 매개변수들을 확인할 수 있음
params = list(net.parameters())
print(len(params))
print(params[0].size()) # conv1's weight
# (3) 화면 출력 확인
```

```
10
torch.Size([6, 1, 5, 5])
```

```
# 다음은 임의의 32 * 32 입력을 가정함
# 참고로 크기가 다른 입력을 받을 때는 입력의 크기를 재조정하거나 신경망 수정함
input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)
# (4) 화면 출력 확인
```

```
tensor([[ 0.1092, -0.0141,  0.1197,  0.0253,  0.0782, -0.0970,  0.0038, -0.1320,
         -0.0988, -0.0248]], grad_fn=<AddmmBackward>)
```

```
# 오류역전파를 통해 그레이디언트를 구하기 전에 모든 가중치의 그레이디언트 버퍼들을 초기화
net.zero_grad()
out.backward(torch.randn(1, 10))

# 손실 함수 정의 및 임의의 값들에 대해서 오차 결과 확인
# nn 패키지는 많이 사용되는 손실함수들을 제공하며, 해당 예제는 단순한 MSE를 사용
output = net(input)
target = torch.randn(10) # a dummy target, for example
target = target.view(1, -1) # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
# (5) 화면 출력 확인
```

```
tensor(1.0218, grad_fn=<MseLossBackward>)
```

```
# 앞에 코드에서 언급한 것과 같이 오류 역전파하기 전, 그레이디언트를 초기화해야 함
# backward() 수행 후 어떤 변화가 있는지 확인하고, 초기화의 필요성을 확인함
net.zero_grad # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)
# (6) 화면 출력 확인
```

```
conv1.bias.grad before backward
tensor([ 0.0296, -0.0171, -0.0220,  0.0220, -0.0109,  0.0105])
```

```
loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
# (7) 화면 출력 확인
```

```
conv1.bias.grad after backward
tensor([ 0.0478, -0.0174, -0.0146,  0.0128, -0.0056,  0.0061])
```

```
# 스토캐스틱 경사 하강법 ((미래) 가중치 = (현재) 가중치 - 학습률 * 그레이디언트)를 이용하
여 가중치 갱신하는 코드는 다음과 같음
learning_rate = 0.01
for f in net.parameters():
```

```
    f.data.sub_(f.grad.data * learning_rate)

# 하지만 위 구현 코드보다 실제, torch.optim에서 구현되는 SDG, Adam, RMSProp 등을 사용함
# 오류 역전파에서 최적화하는 방법을 보인 예제 코드
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad() # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step() # Does the update
```

3. 분류기 학습

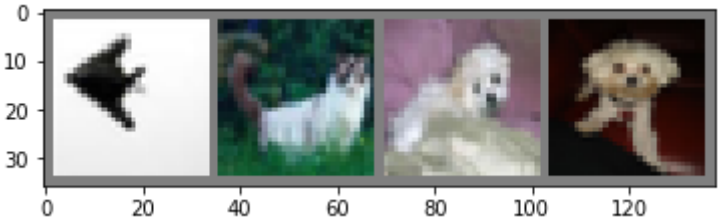
```
### 1. 정규화된 CIFAR-10 훈련집합과 테스트집합을 torchvision을 이용하여 적재함
import torch
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True,
num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=False,
num_workers=2)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck')
# (1) 화면 출력 확인
```

```
Files already downloaded and verified
Files already downloaded and verified
```

```
# 훈련집합의 일부 사진들 확인
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image
def imshow(img):
    img = img / 2 + 0.5 # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1,2,0)))

# get some random training images
dataiter = iter(trainloader)
```

```
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
# (2) 화면 출력 확인
```

```
plane   cat   dog   dog
```




```
### 2. 컨볼루션 신경망을 정의함
# 3채널 32*32 크기의 사진을 입력받고, 신경망을 통과해 10 부류를 수행
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x))) # out : 14 * 14 * 6
        x = self.pool(F.relu(self.conv2(x))) # out : 5 * 5 * 16
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x)) # out : 1 * 1 * 120
        x = F.relu(self.fc2(x)) # out : 1 * 1 * 84
        x = self.fc3(x) # out : 1 * 1 * 10

        return x

net = Net()
# print(net)
```

```
### 3. 손실함수 정의, 교차 엔트로피와 SGD + momentum
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

### 4. 훈련집합을 이용하여 신경망을 학습시킴
for epoch in range(2): # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
```

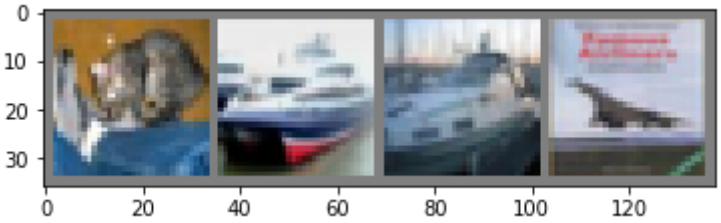
```
        # get the inputs
        inputs, labels = data

        # 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 % 1000 == 999: # print every 1000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                    (epoch + 1, i + 1, running_loss / 1000))
            running_loss = 0.0
print('Finished Training')
# (3) 화면 출력 확인 및 학습이 되고 있는지 서술
```

```
[1,  1000] loss: 2.293
[1,  2000] loss: 2.052
[1,  3000] loss: 1.840
[1,  4000] loss: 1.751
[1,  5000] loss: 1.657
[1,  6000] loss: 1.607
[1,  7000] loss: 1.590
[1,  8000] loss: 1.535
[1,  9000] loss: 1.509
[1, 10000] loss: 1.479
[1, 11000] loss: 1.445
[1, 12000] loss: 1.452
[2,  1000] loss: 1.432
[2,  2000] loss: 1.392
[2,  3000] loss: 1.369
[2,  4000] loss: 1.354
[2,  5000] loss: 1.361
[2,  6000] loss: 1.363
[2,  7000] loss: 1.333
[2,  8000] loss: 1.319
[2,  9000] loss: 1.287
[2, 10000] loss: 1.309
[2, 11000] loss: 1.324
[2, 12000] loss: 1.292
Finished Training
```

- 학습이 진행됨에 따라 손실함수값(loss)이 줄어들고 있는 것으로 보아 학습이 잘 되고 있다

```
### 5. 테스트집합을 이용하여 신경망 성능 확인
dataiter = iter(testloader)
images, labels = dataiter.next()

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth:', ''.join('%5s' % classes[labels[j]] for j in range(4)))
# (4) 화면 출력 확인
```

```
GroundTruth:   cat ship shipplane
```




```
outputs = net(images)
_, predicted = torch.max(outputs, 1)
print('Predicted: ', ''.join('%5s' % classes[predicted[j]] for j in range(4)))
# (5) 화면 출력 확인
```

```
Predicted:    cat  car shipplane
```

```
# performance on the whole test dataset
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (100 * correct
/ total))
# (6) 화면 출력 확인 및 일반화 성능 서술
```

```
Accuracy of the network on the 10000 test images: 53 %
```

- 데이터셋을 2바퀴 돌며 학습한 결과, 신경망이 테스트집합에 대해 53%의 정확도로 분류를 수행하였으며, 괜찮은 일반화 성능을 보였다.

```
# performance on each class
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
```

```
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

for i in range(10):
    print('Accuracy of %5s : %2d %%' % (classes[i], 100 * class_correct[i] /
class_total[i]))

#(7) 화면 출력 확인 및 부류별 분류기의 성능 서술
```

```
Accuracy of plane : 52 %
Accuracy of   car : 70 %
Accuracy of  bird : 39 %
Accuracy of   cat : 54 %
Accuracy of  deer : 39 %
Accuracy of   dog : 37 %
Accuracy of  frog : 57 %
Accuracy of horse : 71 %
Accuracy of  ship : 72 %
Accuracy of truck : 44 %
```

- 부류별 분류기의 성능이 서로 다른 모습인데, 더 많은 학습을 통해 성능을 높일 수 있을 것이다.

4.

컨볼루션 층의 입력 크기가 $32 * 32 * 3$이고

- (a) 10개 $5 * 5$ 필터들을 보폭 1과 덧대기 2로 적용하였을 때 출력의 크기와 매개변수의 수를 구하세요

  출력의 크기 = $W2 * H2 * D2$

  $W2 = (W1 - F + 2P)/S + 1 = (32 - 5 + 4)/1 + 1 = 32$

  $H2 = (H1 - F + 2P)/S + 1 = (32 - 5 + 4)/1 + 1 = 32$

  $D2 = K = 10$

  따라서, 출력의 크기 = $32 * 32 * 10$

  매개변수의 수 = $(F * F * D1)K + K = (5 * 5 * 3)10 + 10 = 760$

- (b) 동일한 입력에 64개 3*3 필터들을 보폭 1과 덧대기 1로 적용하였을 때 출력의 크기와 매개변수의 수

  출력의 크기 = $W2 * H2 * D2$

  $W2 = (W1 - F + 2P)/S + 1 = (32 - 3 + 2)/1 + 1 = 32$

  $H2 = (H1 - F + 2P)/S + 1 = (32 - 3 + 2)/1 + 1 = 32$

  $D2 = K = 64$

  따라서, 출력의 크기 = $32 * 32 * 64$

  매개변수의 수 = $(F * F * D1)K + K = (3 * 3 * 3)64 + 64 = 1792$

5.

다음 조건을 만족하는 컨볼루션 신경망을 구현하고, 3번의 (3), (6), (7)의 성능결과를 확인하고 비교하세요

- (1) INPUT - CONV(32 3*3) - CONV(32 3*3) - RELU - POOL - CONV(32 3*3) - CONV(32 3*3) - RELU - POOL - FC - OUTPUT

```
# (1)
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 32, 3)
        self.conv3 = nn.Conv2d(32, 32, 3)
        self.conv4 = nn.Conv2d(32, 32, 3)
        self.fc1 = nn.Linear(32 * 5 * 5, 10)

    def forward(self, x):
        x = self.conv1(x) # 30 * 30 * 32
        x = self.pool(F.relu(self.conv2(x))) # 14 * 14 * 32
        x = self.conv3(x) # 12 * 12 * 32
        x = self.pool(F.relu(self.conv4(x))) # 5 * 5 * 32
        x = x.view(-1, 32*5*5)
        x = self.fc1(x)

        return x

net = Net()
```

```
[1,  1000] loss: 2.171
[1,  2000] loss: 1.842
[1,  3000] loss: 1.676
[1,  4000] loss: 1.552
[1,  5000] loss: 1.492
[1,  6000] loss: 1.419
[1,  7000] loss: 1.353
[1,  8000] loss: 1.310
[1,  9000] loss: 1.286
[1, 10000] loss: 1.262
[1, 11000] loss: 1.225
[1, 12000] loss: 1.192
[2,  1000] loss: 1.141
[2,  2000] loss: 1.137
[2,  3000] loss: 1.103
[2,  4000] loss: 1.083
[2,  5000] loss: 1.095
[2,  6000] loss: 1.104
[2,  7000] loss: 1.065
[2,  8000] loss: 1.070
[2,  9000] loss: 1.062
[2, 10000] loss: 1.065
[2, 11000] loss: 1.063
[2, 12000] loss: 1.019
Finished Training
```

```
Accuracy of the network on the 10000 test images: 63 %
```

```
Accuracy of plane : 71 %
Accuracy of   car : 81 %
Accuracy of  bird : 50 %
Accuracy of   cat : 65 %
Accuracy of  deer : 58 %
Accuracy of   dog : 32 %
Accuracy of  frog : 55 %
Accuracy of horse : 62 %
Accuracy of  ship : 77 %
Accuracy of truck : 77 %
```

    - 기존의 신경망보다 조금 더 좋은 성능을 보였다
- (2) 3번 문제의 신경망에 Adam 최적화 (강의자료의 기본 hyper-parameters 사용) 적용

```
### 3. 손실함수 정의, 교차 엔트로피와 SGD + momentum
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
# Adam 최적화 - 일반적으로α1=0.9, α2=0.999, ρ=1e-3 혹은 5e-4 설정
optimzer = optim.Adam(net.parameters(),lr=1e-3, betas=(0.9,0.999))
```

```
[1,  1000] loss: 2.307
[1,  2000] loss: 2.305
[1,  3000] loss: 2.305
[1,  4000] loss: 2.305
[1,  5000] loss: 2.306
[1,  6000] loss: 2.304
[1,  7000] loss: 2.306
[1,  8000] loss: 2.308
[1,  9000] loss: 2.307
[1, 10000] loss: 2.307
[1, 11000] loss: 2.305
[1, 12000] loss: 2.305
[2,  1000] loss: 2.306
[2,  2000] loss: 2.306
[2,  3000] loss: 2.307
[2,  4000] loss: 2.304
[2,  5000] loss: 2.306
[2,  6000] loss: 2.307
[2,  7000] loss: 2.303
[2,  8000] loss: 2.309
[2,  9000] loss: 2.305
[2, 10000] loss: 2.307
[2, 11000] loss: 2.306
[2, 12000] loss: 2.305
Finished Training
```

```
Accuracy of the network on the 10000 test images: 7 %
```

```
Accuracy of plane :  0 %
Accuracy of   car : 23 %
Accuracy of  bird :  0 %
Accuracy of   cat :  0 %
Accuracy of  deer : 29 %
Accuracy of   dog :  0 %
Accuracy of  frog : 24 %
Accuracy of horse :  0 %
Accuracy of  ship :  0 %
Accuracy of truck :  0 %
```

    - Adam 최적화 결과 기존 신경망에 비해 일반화 성능이 현저히 떨어졌다
- (3) 데이터 확대 방법들 중 하나를 적용한 후, 3번 문제의 신경망 학습 (Hint : transforms)

```
### 1. 정규화된 CIFAR-10 훈련집합과 테스트집합을 torchvision을 이용하여 적재함
import torch
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ColorJitter(hue=(-0.3,0.3)),
transforms.ToTensor(), transforms.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5))])
```

```
[1,  1000] loss: 2.293
[1,  2000] loss: 2.144
[1,  3000] loss: 2.009
[1,  4000] loss: 1.890
[1,  5000] loss: 1.806
[1,  6000] loss: 1.761
[1,  7000] loss: 1.671
[1,  8000] loss: 1.667
[1,  9000] loss: 1.630
[1, 10000] loss: 1.577
[1, 11000] loss: 1.571
[1, 12000] loss: 1.534
[2,  1000] loss: 1.489
[2,  2000] loss: 1.507
[2,  3000] loss: 1.430
[2,  4000] loss: 1.436
[2,  5000] loss: 1.421
[2,  6000] loss: 1.423
[2,  7000] loss: 1.373
[2,  8000] loss: 1.385
[2,  9000] loss: 1.349
[2, 10000] loss: 1.367
[2, 11000] loss: 1.333
[2, 12000] loss: 1.329
Finished Training
```

```
Accuracy of the network on the 10000 test images: 50 %
```

```
Accuracy of plane : 42 %
Accuracy of   car : 52 %
Accuracy of  bird : 48 %
Accuracy of   cat : 44 %
Accuracy of  deer : 34 %
Accuracy of   dog : 34 %
Accuracy of  frog : 61 %
Accuracy of horse : 43 %
Accuracy of  ship : 68 %
Accuracy of truck : 74 %
```

    - `ColorJitter(hue=(-0.3,0.3))`를 적용한 결과 기존 신경망과 비슷한 결과가 나왔다
- (4) 3번 문제의 신경망에 CONV 층마다 배치 정규화를 적용 (Hint : nn.BatchNorm)

```python
### 2. 컨볼루션 신경망을 정의함
# 3채널 32*32 크기의 사진을 입력받고, 신경망을 통과해 10 부류를 수행
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        # (4)
        self.batchnorm1 = nn.BatchNorm2d(6)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # (4)
        self.batchnorm2 = nn.BatchNorm2d(16)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.batchnorm1(self.conv1(x))))
        x = self.pool(F.relu(self.batchnorm2(self.conv2(x))))
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x

net = Net()
# print(net)
```

```
[1,  1000] loss: 2.093
[1,  2000] loss: 1.847
[1,  3000] loss: 1.725
[1,  4000] loss: 1.710
[1,  5000] loss: 1.662
[1,  6000] loss: 1.604
[1,  7000] loss: 1.581
[1,  8000] loss: 1.532
[1,  9000] loss: 1.532
[1, 10000] loss: 1.505
```

```
[1, 11000] loss: 1.470
[1, 12000] loss: 1.460
[2,  1000] loss: 1.426
[2,  2000] loss: 1.406
[2,  3000] loss: 1.377
[2,  4000] loss: 1.374
[2,  5000] loss: 1.379
[2,  6000] loss: 1.363
[2,  7000] loss: 1.362
[2,  8000] loss: 1.385
[2,  9000] loss: 1.337
[2, 10000] loss: 1.332
[2, 11000] loss: 1.309
[2, 12000] loss: 1.320
Finished Training
```

```
Accuracy of the network on the 10000 test images: 52 %
```

```
Accuracy of plane : 66 %
Accuracy of   car : 55 %
Accuracy of  bird : 47 %
Accuracy of   cat : 35 %
Accuracy of  deer : 42 %
Accuracy of   dog : 23 %
Accuracy of  frog : 69 %
Accuracy of horse : 61 %
Accuracy of  ship : 53 %
Accuracy of truck : 71 %
```

    - 컨볼루션층에 배치 놈 적용 결과 기존 신경망과 비슷한 일반화 성능이 나왔다

- (5) 3번 문제의 신경망에 로그우도 (-log) 손실함수를 적용

```
### 2. 컨볼루션 신경망을 정의함
# 3채널 32*32 크기의 사진을 입력받고, 신경망을 통과해 10 부류를 수행
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        x = F.log_softmax(x)
```

```
        return x

net = Net()
# print(net)

### 3. 손실함수 정의, 교차 엔트로피와 SGD + momentum
import torch.optim as optim

criterion = nn.NLLLoss2d()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

```
[1,  1000] loss: 2.299
[1,  2000] loss: 2.176
[1,  3000] loss: 1.928
[1,  4000] loss: 1.785
[1,  5000] loss: 1.690
[1,  6000] loss: 1.608
[1,  7000] loss: 1.571
[1,  8000] loss: 1.569
[1,  9000] loss: 1.477
[1, 10000] loss: 1.513
[1, 11000] loss: 1.433
[1, 12000] loss: 1.451
[2,  1000] loss: 1.400
[2,  2000] loss: 1.382
[2,  3000] loss: 1.368
[2,  4000] loss: 1.345
[2,  5000] loss: 1.352
[2,  6000] loss: 1.312
[2,  7000] loss: 1.345
[2,  8000] loss: 1.317
[2,  9000] loss: 1.265
[2, 10000] loss: 1.286
[2, 11000] loss: 1.263
[2, 12000] loss: 1.223
Finished Training
```

```
Accuracy of the network on the 10000 test images: 56 %
```

```
Accuracy of plane : 59 %
Accuracy of   car : 71 %
Accuracy of  bird : 45 %
Accuracy of   cat : 39 %
Accuracy of  deer : 54 %
Accuracy of   dog : 41 %
Accuracy of  frog : 57 %
Accuracy of horse : 68 %
Accuracy of  ship : 76 %
Accuracy of truck : 49 %
```

    - 로그우도 손실함수 적용 결과 기존 신경망보다 약간 더 나은 일반화 성능을 보였다

- (6) 3번 문제의 신경망에 L2놈 규제 적용

```
### 3. 손실함수 정의, 교차 엔트로피와 SGD + momentum
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9,
weight_decay=2)
```

```
[1,  1000] loss: 2.303
[1,  2000] loss: 2.303
[1,  3000] loss: 2.303
[1,  4000] loss: 2.303
[1,  5000] loss: 2.303
[1,  6000] loss: 2.303
[1,  7000] loss: 2.303
[1,  8000] loss: 2.303
[1,  9000] loss: 2.303
[1, 10000] loss: 2.303
[1, 11000] loss: 2.303
[1, 12000] loss: 2.303
[2,  1000] loss: 2.302
[2,  2000] loss: 2.303
[2,  3000] loss: 2.303
[2,  4000] loss: 2.303
[2,  5000] loss: 2.303
[2,  6000] loss: 2.303
[2,  7000] loss: 2.303
[2,  8000] loss: 2.303
[2,  9000] loss: 2.303
[2, 10000] loss: 2.303
[2, 11000] loss: 2.303
[2, 12000] loss: 2.303
Finished Training
```

```
Accuracy of the network on the 10000 test images: 10 %
```

```
Accuracy of plane :  0 %
Accuracy of   car :  0 %
Accuracy of  bird :  0 %
Accuracy of   cat :  0 %
Accuracy of  deer :  0 %
Accuracy of   dog :  0 %
Accuracy of  frog :  0 %
Accuracy of horse : 100 %
Accuracy of  ship :  0 %
Accuracy of truck :  0 %
```

- $\lambda = 2$로 학습을 진행하였는데, 손실함수가 줄어들지 않아 기존 신경망에 비해 현저히 낮은 일반화 성능을 보였다
- 모든 데이터를 horse로 분류한 것 같다

6.

신경망의 출력이 $(0.4, 2.0, 0.001, 0.32)^T$일 때 소프트맥스 함수를 적용한 결과를 쓰시오

- $(0.1325, 0.6563, 0.0889, 0.1223)^T$ (소수점 넷째자리로 반올림)

7.

소프트맥스 함수를 적용한 후 출력이 $(0.001, 0.9, 0.001, 0.098)^T$ 이고 레이블 정보가 $(0, 0, 0, 1)^T$ 일때, 세 가지 목적함수, 평균제곱 오차, 교차 엔트로피, 로그우도를 계산하시오

- 평균제곱오차

  $e = \frac{1}{n}\sum_{i=1}^{n}(y_i - t_i)^2$

  $= \frac{1}{4}((0.001-0)^2 + (0.9-0)^2 + (0.001-0)^2 + (0.098-1)^2)$

  $= 0.4059$ (소수점 넷째자리로 반올림)

- 교차 엔트로피

  $e = -\sum_{i=1}^{n}(y_i log_2 o_i + (1-y_i) log_2 (1-o_i))$

  $= -(log_2(1-0.001) + log_2(1-0.9) + log_2(1-0.001) + log_2(0.098))$

  $= 6.6759$ (소수점 넷째자리로 반올림)

- 로그우도

  $e = -log_2 o_y$

  $= -log_2(0.098)$

  $= 3.3511$ (소수점 넷째자리로 반올림)