인공지능3

20130940 장용훈

```
#1-1
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution 커널사이즈
        self.conv1 = nn.Conv2d(1, 6, 5)
        # conv1의 결과과 conv2로 들어가기 때문에 input = 6,
        self.conv2 = nn.Conv2d(6, 16, 5)
        # y = Wx + b, 연산을 수행 3번의 수행을 거쳐서 마지막  특징값의 크기는 10  Linear을 연산을 3번 수행함으로
        # 차원의 축소, 즉 최종적으로 10크기를 얻음.
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling 이때, 데이터에 relu함수를 적용하고 (2, 2) window를 이용하여 차원축소
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # 위와 똑같은 기능을 한다. 다만 다른 크기로 적용, (2,2) 와 2는 똑같이 2x2 window를 사용
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        # row의 수는 col 안다면 자동으로 계산. col 의 수눈 num_flat_featrues를 통해 구함. 16*5*5를 얻음
        # 즉 연산을 하기위해 데이터의 차원을 조정
        x = x.view(-1, self.num_flat_features(x))
        # y = Wx + b한 값을 relu함수로 적용
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        # 10 크기의 값이 구해진다.
        x = self.fc3(x)
        return x
```

```
    def num_flat_features(self, x):
        # 여기서  col의 크기를 얻음. 16*5*5를 얻기 위해. 마지막 conv에 따라 값이 정해진다.
        size = x.size()[1:]
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
# 위에서 정의한 CNN 모델을 출력한다.
print(net)
# conv1 과 conv2 의 컨볼루션층을 통과하며 학습을 진행한 다음
# fc를 통과한다. 이 때  y = Wx + b 을 각각 수행 한다. 최종적으로 10개의 클래스로 분류가 된다.
# 층이 여러개여서, 그래디언트 소멸 문제가 발생하는데 이는, relu함수로 이를 감소 시킨다.
# 그리거 Max Pooling 을 이용해 feature 를 요약한다.
```

```
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)
)
```

```
#1-2
# 모듈 매개 변수로 간주되는 일종의 Tensor 자동으로 매개 변수 목록에 추가
params = list(net.parameters())
print(len(params))
# conv1의 weight 값
print(params[0].size())
```

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

```
# 1-3
# net으로 정의한 cnn에 랜덤값 적용
input = torch.randn(1,1,32,32)
out = net(input)
print(out)
```

```
tensor([[ 0.0844, -0.0203,  0.0444,  0.1107,  0.0257, -0.0911,  0.0968, -0.0900,
          0.1139,  0.0394]], grad_fn=<ThAddmmBackward>)
```

```
# 1-4
# gradient를 초기화
net.zero_grad()
# backward 연산 수행
out.backward(torch.randn(1,10))
output = net(input)
target = torch.randn(10)
# output 과 같은 사이즈로 만들어 준다.
target = target.view(1,-1)
criterion = nn.MSELoss()
loss = criterion(output,target)
# mse를 적용한 loss 값을 얻음
print(loss)
```

```
tensor([[ 0.0844, -0.0203,  0.0444,  0.1107,  0.0257, -0.0911,  0.0968, -0.0900,
          0.1139,  0.0394]], grad_fn=<ThAddmmBackward>)
tensor(0.6251, grad_fn=<MseLossBackward>)
```

```
# 1-5
# gradient 초기화
net.zero_grad()
print("conv1.bias.grad before backward")
# conv1의 bias gradient 출력 초기화 했기에 0이나옴
print(net.conv1.bias.grad)
```

```
conv1.bias.grad before backward
tensor([0., 0., 0., 0., 0., 0.])
```

```
# 1-6
loss.backward()
print('conv1.bias.grad after backward')
# conv1의 bias gradient 출력, backward 연산을 수행 했기에 0이 아닌 다른 값이 나옴
print(net.conv1.bias.grad)
```

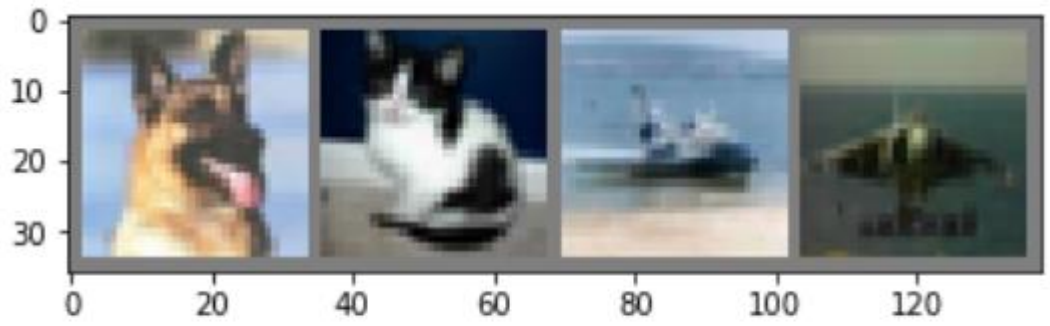
```
conv1.bias.grad after backward
tensor([ 0.0009,  0.0184,  0.0547, -0.1052, -0.0111,  0.0556])
```

```
# 1-7
# SGD를 이용한 가중치 갱신
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
import torch.optim as optim
# SGD 뿐만 아니라 다향한 알고리즘을 사용가능 이때, optim를 활용.
optimizer = optim.SGD(net.parameters(),lr=0.01)
# gradient 초기화
optimizer.zero_grad()
output = net(input)
# 위의 가중치를 구하는 원리는 동일하지만, 방법은 다른 방법 optim을 사용.
loss = criterion(output,target)
loss.backward()
optimizer.step()
```

```
# 2-1
# 데이터를 변형시킬 때 사용, list를 tensor로 만들어주고 여기에 정규화 진행.
# Compose 는 여러 transform 들을 chaining, 즉 여러 transform을 진행.
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# CIFAR10 dataset에서 data를 다운 받아옴 이는 훈련집합
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
# 훈련집합에서 4개의 데이터를 불러온다. 이때 데이터를 섞어서 불러온다, 2개의 하위 프로세스 사용
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)
# 검증집합을 정의 _CIFAR10에서 데이터를 받아옴.
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
# 검증집합에서 4개의 데이터를 불러온다, 2개의 하위 프로세스 사용
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')
```

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

```python
# 2-2
# 이미지를 출력하기위해 정의한 함수, 이때 정규화 하지 않는다.
def imshow(img):
# 정규화한 이미지를 원복
    img = img / 2 + 0.5
    npimg = img.numpy()
# 출력은 하지 않고, 이미지를 그려주는 역할 이때 RGB의 순서를 변경하여 출력.
    plt.imshow(np.transpose(npimg, (1, 2, 0)))


# 랜덤으로 이미지를 가져온다.
dataiter = iter(trainloader)
images, labels = dataiter.next()

# 이미지를 출력한다.
imshow(torchvision.utils.make_grid(images))
plt.show()
# 어떤 label 있는지 출력
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```




```
dog   cat plane plane
```

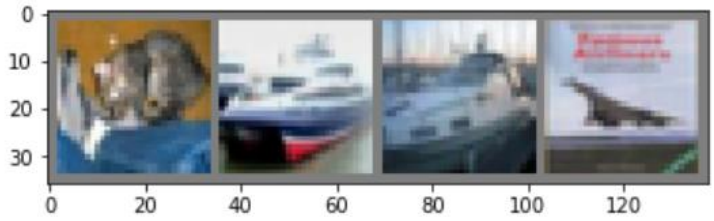
```python
# 2-3
# CNN모델을 정의
# 3채널 32 * 32 크기의 사진을 입력받아, 신경망을 통과해 10개의 class로 분류
# 1번에서 정의한 모델과 같은 동작. in-conv-relu-maxpool--conv-relu-maxpool-fc-fcfc-out 의 구조를 가짐
class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()
        # 3 input image channel, 6 output channels, 5x5 square convolution 커널사이즈
        self.conv1 = nn.Conv2d(3, 6, 5)
        # pooling은 2x2
        self.pool = nn.MaxPool2d(2, 2)
        # 6 input image channel, 16 output channels, 5x5 square convolution 커널사이즈
        self.conv2 = nn.Conv2d(6, 16, 5)
        # y = Wx + b, 연산을 수행 3번의 수행을 거쳐서 마지막  특징값의 크기는 10  Linear을 연산을 3번 수행함으로
        # 차원의 축소, 즉 최종적으로 10크기를 얻음.
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    # foward를 정의
```

```
    def forward(self, x):
        # conv1구조로 pooling 이때 relu 함수를 적용
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        # row의 수는 col 안다면 자동으로 계산. col 의 수눈 num_flat_featrues를 통해 구함. 16*5*5를 얻음
        # 즉 연산을 하기위해 데이터의 차원을 조정
        x = x.view(-1, 16 * 5 * 5)
        # y = Wx + b한 값을 relu함수로 적용
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        # 10 크기의 값이 구해진다.
        x = self.fc3(x)
        return x

net = Net2()
```

```
# 2-4
# 위에서 정의한 모델로 학습을 시킨다.
# loss함수로 croosentropy사용
criterion = nn.CrossEntropyLoss()
# SGD 적용, 학습률= 0.001, 모멘텀은 0.9(수렴 속도 향상 효과)
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum = 0.9)
for epoch in range(2):
    # 학습이 진행되는지 알기위헤 0으로 초기화
    running_loss = 0.0
    for i, data in enumerate(trainloader,0):
        inputs,labels = data
        # gradient 초기화
        optimizer.zero_grad()
        # 정의한 모델로 학습
        outputs = net(inputs)
        # 손실값
        loss = criterion(outputs, labels)
        # backward 진행
        loss.backward()
        # optimiezer 진행
        optimizer.step()

        running_loss += loss.item()
        # 천개 단위로 손실이 어떻게 나타나는 지 출력, 즉 12000개 샘플에 관한 표시
        if i % 1000 == 999:
            print("[%d, %5d] loss: %.3f" % (epoch+1,i+1,running_loss/1000))
            running_loss = 0.0
print('Finished Training')
# epoch가 수행 될때 마다 loss가 2.293에서 1.248 까지로 떨어진 것이 보인다
# 오류가 감소했기 때문에 적절한 방향으로 학습이 되었다.
```

```
[1,  1000] loss: 2.296
[1,  2000] loss: 2.096
[1,  3000] loss: 1.899
[1,  4000] loss: 1.778
[1,  5000] loss: 1.694
[1,  6000] loss: 1.635
[1,  7000] loss: 1.599
[1,  8000] loss: 1.540
[1,  9000] loss: 1.518
[1, 10000] loss: 1.507
[1, 11000] loss: 1.490
[1, 12000] loss: 1.457
[2,  1000] loss: 1.412
[2,  2000] loss: 1.388
[2,  3000] loss: 1.357
[2,  4000] loss: 1.369
[2,  5000] loss: 1.372
[2,  6000] loss: 1.349
[2,  7000] loss: 1.344
[2,  8000] loss: 1.314
[2,  9000] loss: 1.309
[2, 10000] loss: 1.281
[2, 11000] loss: 1.285
[2, 12000] loss: 1.296
Finished Training
```

```
# 2-4
# 학습이 제대로 되어있는지에 대한 검증을 위한 정의
dataiter = iter(testloader)
images, labels = dataiter.next()

# 검증 데이터가 제대로 되었는지에 관한 출력
imshow(torchvision.utils.make_grid(images))
plt.show()
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
```




```
GroundTruth:    cat  ship  ship plane
```

```
# 2-5 학습결과가에 관한
# 모델이 이미지에 관하여 어떻게 평가하는지에 대한 출력
outputs = net(images)
# 예측 결과, output은 모델에 대한 예측값이다.(croosEntropyLoss를 사용했기 때문)
_, predicted = torch.max(outputs, 1)
# 얻은예측이 실제 어떤레이블로 나타내는지 출력
print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```

```
Predicted:   frog  ship   car  bird
```

```
# 2-6
# 검증 시작
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        # data는 이미지와 라벨로 구성
        images, labels = data
        # images를 학습
        outputs = net(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs.data,1)
        total += labels.size(0)
        # 정답 개수 확인
        correct += (predicted == labels).sum().item()
# 정답률 출력
print('Accuracy of the network on the 10000 test images: %d %%' % (100*correct/total))
# 정의한 모델에서 Accuracy는 50% ~ 60% 의 결과가 나타난다.
# 따라서 좋은 일반화 성능을 보이지는 않는다. 다만 규제를 이용하거나 모델에 층을 더 깊게쌓는 것들과 같은 방법으로
# 성능을 높일 수 있는 여지는 남아있다.
```

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

```python
# 2-7
# 이번에는 10개의 라벨에 관한 정답률 확인
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:
         # data는 이미지와 라벨로 구성
        images, labels = data
        # image 학습
        outputs = net(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs, 1)
        # squeeze() = 차원의 사이즈가 '1'인 차원을 제거. 여기서는 squeeze 를 호출하지 않아도 동일한 결과가 나옴.
        c = (predicted == labels).squeeze()
        # batchsize가 4이기 때문에.
        for i in range(4):
            label = labels[i]
            # label 과 predict가 일치한 경우를 더함
            class_correct[label] += c[i].item()
            # 한 라벨에 몇개를 검증 했는지에 대한 값
            class_total[label] += 1
```

```python
# 결과
for i in range(10):
    print('Accuracy of %5s : %2d %%' %(classes[i], 100 * class_correct[i] / class_total[i]))
# 일반화의 ACCURACY가 50~60퍼센트라는 낮은 성능을 보이기에 부류별 분류기의 성능 또한 전반적으로 낮은 값을 보이게 된다
# 그리고 각 부분마다 ACUURACY의 차이가 큰것으로 보아 모델에 문제가 있음을 알 수 있다
# 좀더 자세히 들여다보면, plane, car, ship, turuck 과 같은 기계류(무생물) 집합은 평균적으로 적중률이 더 높다
# 그러나 bird,cat,derr,dog,frog,hose 생물 집합은 평균적으로 더 낮은 적중률을 보인다
# 하지만 이는 절대적이지는 않다. frog의 적중률이 가장 높기 때문이다.
# cat은 적중률이 심각하게 낮다.
# 따라서 성능이 좋지는 않다고 평가 할 수 있다.
```

```
Accuracy of plane : 43 %
Accuracy of   car : 76 %
Accuracy of  bird : 65 %
Accuracy of   cat : 12 %
Accuracy of  deer : 35 %
Accuracy of   dog : 48 %
Accuracy of  frog : 79 %
Accuracy of horse : 38 %
Accuracy of  ship : 61 %
Accuracy of truck : 64 %
```

```python
# 3-1 컨볼루션 모델 변경
# CNN모델을 정의
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 3 input image channel, 32 output channels, 3x3 square convolution 커널사이즈
        self.conv1 = nn.Conv2d(3, 32, 3)
        # pooling은 2x2
        self.pool = nn.MaxPool2d(2, 2)
        # 32 input image channel, 32 output channels, 3x3 square convolution 커널사이즈
        self.conv2 = nn.Conv2d(32, 32, 3)
        self.conv3 = nn.Conv2d(32, 32, 3)
        self.conv4 = nn.Conv2d(32, 32, 3)
        # y = Wx + b, 연산을 수행 3번의 수행을 거쳐서 마지막  특징값의 크기는 10  Linear을 연산을 3번 수행함으로
        # 차원의 축소, 즉 최종적으로 10크기를 얻음.
        self.fc1 = nn.Linear(32 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

```
    # foward를 정의
    def forward(self, x):
        # conv1구조로 pooling 이때 relu 함수를 적용
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.pool(F.relu(x))
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.pool(F.relu(x))
        # row의 수는 col 안다면 자동으로 계산. col 의 수눈 num_flat_featrues를 통해 구함. 16*5*5를 얻음
        # 즉 연산을 하기위해 데이터의 차원을 조정
        x = x.view(-1, 32 * 5 * 5)
        # y = Wx + b한 값을 relu함수로 적용
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        # 10 크기의 값이 구해진다.
        x = self.fc3(x)
        return x

net1 = Net()
# 2번 모델과 비교했을 때, 더 깊이(2개의 conv층 추가) 쌓은 것이 특징이다.
```

```
# 3-1(3)
import torch.optim as optim

# loss함수로 croosentropy사용
criterion = nn.CrossEntropyLoss()
# SGD 적용, 학습률= 0.001, 모멘텀은 0.9(수렴 속도 향상 효과)
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum = 0.9)
for epoch in range(2):
    # 학습이 진행되는지 알기위헤 0으로 초기화
    running_loss = 0.0
    for i, data in enumerate(trainloader,0):
        inputs,labels = data
        # gradient 초기화
        optimizer.zero_grad()
        # 정의한 모델로 학습
        outputs = net1(inputs)
        # 손실값
        loss = criterion(outputs, labels)
        # backward 진행
        loss.backward()
        # optimiezer 진행
        optimizer.step()
```

```
        running_loss += loss.item()
        # 천개 단위로 손실이 어떻게 나타나는 지 출력, 즉 12000개 샘플에 관한 표시
        if i % 1000 == 999:
            print("[%d, %5d] loss: %.3f" % (epoch+1,i+1,running_loss/1000))
            running_loss = 0.0
print('Finished Training')
# 2번에서 최종값은 1.296이었고 현재 모델의 최종값은 1.075이다. 왜냐하면
# 2번과 비교 했을 때,더 깊이 쌓았고 3*3이라는 더작은 커널을 이용했기 때문에
# 성능이 더 좋음을 알 수 있다.
```

```
[1,  1000] loss: 2.298
[1,  2000] loss: 2.123
[1,  3000] loss: 1.985
[1,  4000] loss: 1.885
[1,  5000] loss: 1.747
[1,  6000] loss: 1.679
[1,  7000] loss: 1.625
[1,  8000] loss: 1.517
[1,  9000] loss: 1.482
[1, 10000] loss: 1.430
[1, 11000] loss: 1.384
[1, 12000] loss: 1.355
[2,  1000] loss: 1.306
[2,  2000] loss: 1.292
[2,  3000] loss: 1.239
[2,  4000] loss: 1.224
[2,  5000] loss: 1.178
[2,  6000] loss: 1.172
[2,  7000] loss: 1.127
[2,  8000] loss: 1.130
[2,  9000] loss: 1.091
[2, 10000] loss: 1.108
[2, 11000] loss: 1.046
[2, 12000] loss: 1.075
Finished Training
```

```python
# 3-1(6)
# 검증 시작
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        # data는 이미지와 라벨로 구성
        images, labels = data
        # images를 학습
        outputs = net1(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs.data,1)
        total += labels.size(0)
        # 정답 개수 확인
        correct += (predicted == labels).sum().item()
# 정답률 출력
print('Accuracy of the network on the 10000 test images: %d %%' % (100*correct/total))
# ACCURACY 역시 기존의 52% 에서 63%으로 상승한 것을 볼 수 있다
# 따라서 이 또한 성능이 개선되었음을 나타낸다.
# 하지만 성능을 높일 수 있는 다른 방법들은 적용하지 않았기에 성능 향상을 노릴 수 있다.
```

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

```python
# 3-1(7)
# 이번에는 10개의 라벨에 관한 정답률 확인
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:
        # data는 이미지와 라벨로 구성
        images, labels = data
        # image 학습
        outputs = net1(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs, 1)
        # squeeze() = 차원의 사이즈가 '1'인 차원을 제거. 여기서는 squeeze 를 호출하지 않아도 동일한 결과가 나옴.
        c = (predicted == labels).squeeze()
        # batchsize가 4이기 때문에.
        for i in range(4):
            label = labels[i]
            # label 과 predict가 일치한 경우를 더함
            class_correct[label] += c[i].item()
            # 한 라벨에 몇개를 검증 했는지에 대한 값
            class_total[label] += 1
# 결과
for i in range(10):
    print('Accuracy of %5s : %2d %%' %(classes[i], 100 * class_correct[i] / class_total[i]))
# 마찬가지로 2번과 유사한 양상을 보인다. 하지만 생물과 기계류의 평균적인 ACCURACY차이는 많이 감소 했음이 나타난다.
# 그러나 여전히 특정 개체 DOG 와 같은 이미지는 잘 맞추지 못한다.
# 특정 부류는 비교적 우수한 성능을 보이지만 특정부류 문제에서는 성능이 낮다. 이는 2번과 동일하다.
```

```
Accuracy of plane : 79 %
Accuracy of   car : 78 %
Accuracy of  bird : 53 %
Accuracy of   cat : 55 %
Accuracy of  deer : 58 %
Accuracy of   dog : 28 %
Accuracy of  frog : 62 %
Accuracy of horse : 62 %
Accuracy of  ship : 83 %
Accuracy of truck : 70 %
```

```python
#3-2(3) ADAM 최적화
# loss함수로 croosentropy사용
criterion = nn.CrossEntropyLoss()
# ADAM 적용, 학습률= 0.001
optimizer = optim.Adam(net.parameters(), lr=0.001)
for epoch in range(2):
    # 학습이 진행되는지 알기위헤 0으로 초기화
    running_loss = 0.0
    for i, data in enumerate(trainloader,0):
        inputs,labels = data
        # gradient 초기화
        optimizer.zero_grad()
        # 정의한 모델로 학습
        outputs = net(inputs)
        # 손실값
        loss = criterion(outputs, labels)
        # backward 진행
        loss.backward()
        # optimiezer 진행
        optimizer.step()

        running_loss += loss.item()
        # 천개 단위로 손실이 어떻게 나타나는 지 출력, 즉 12000개 샘플에 관한 표시
        if i % 1000 == 999:
            print("[%d, %5d] loss: %.3f" % (epoch+1,i+1,running_loss/1000))
            running_loss = 0.0
print('Finished Training')
```

```
# 2번과 다른점은 LOSS가 2 -> 1.752로 확 내려가는 모습이 나타난다. loss 변화의 폭이 더 크다. 즉 학습이 더빠르다.
# loss가 감소하는 방향임으로 적절하게 학습이 되고 있다.
# 이전에는 2.296 -> 2.096 -> 1.899 나타나는 loss 변화의 폭이 더 작다.
# 그럼에도 불구하고 최종적인 결과는 1.236인데, 이는 2번의 결과 값 1.296과 큰차이가 나지 않는다.
# 이로 짐작했을 때, optimizer의 적용 알고리즘만의 변경만 으로는 커다란 성능 향상을 기대 할 수 없다.
```

```
[1,  1000] loss: 2.007
[1,  2000] loss: 1.752
[1,  3000] loss: 1.603
[1,  4000] loss: 1.594
[1,  5000] loss: 1.548
[1,  6000] loss: 1.501
[1,  7000] loss: 1.471
[1,  8000] loss: 1.483
[1,  9000] loss: 1.425
[1, 10000] loss: 1.416
[1, 11000] loss: 1.411
[1, 12000] loss: 1.359
[2,  1000] loss: 1.333
[2,  2000] loss: 1.308
[2,  3000] loss: 1.317
[2,  4000] loss: 1.267
[2,  5000] loss: 1.258
[2,  6000] loss: 1.278
[2,  7000] loss: 1.253
[2,  8000] loss: 1.225
[2,  9000] loss: 1.242
[2, 10000] loss: 1.235
[2, 11000] loss: 1.253
[2, 12000] loss: 1.236
Finished Training
```

```
# 3-2(6)
# 검증 시작
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        # data는 이미지와 라벨로 구성
        images, labels = data
        # images를 학습
        outputs = net(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs.data,1)
        total += labels.size(0)
        # 정답 개수 확인
        correct += (predicted == labels).sum().item()
# 정답률 출력
print('Accuracy of the network on the 10000 test images: %d %%' % (100*correct/total))
# ACCURACY 역시 기존의 52% 에서 54%으로 상승한 것을 볼 수 있다
# 성능이 향상되기는 했지만, 엄청 큰차이가 나지는 않는다
# 따라서 OPTIMIZER의 변경 만으로는 큰 성능 향상을 나타낼 수 없음을 다시한번 확인 했다.
```

```
Accuracy of the network on the 10000 test images: 54 %
```

```
# 3-2(7)
# 이번에는 10개의 라벨에 관한 정답률 확인
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:
        # data는 이미지와 라벨로 구성
        images, labels = data
        # image 학습
        outputs = net(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs, 1)
        # squeeze() = 차원의 사이즈가 '1'인 차원을 제거. 여기서는 squeeze 를 호출하지 않아도 동일한 결과가 나옴.
        c = (predicted == labels).squeeze()
        # batchsize가 4이기 때문에.
        for i in range(4):
            label = labels[i]
            # label 과 predict가 일치한 경우를 더함
            class_correct[label] += c[i].item()
            # 한 라벨에 몇개를 검증 했는지에 대한 값
            class_total[label] += 1
```

```
# 결과
for i in range(10):
    print('Accuracy of %5s : %2d %%' %(classes[i], 100 * class_correct[i] / class_total[i]))
# 특정 부류는 비교적 우수한 성능을 보이지만 특정부류 문제에서는 성능이 낮다. (이는 2번과 동일하다.)
# 2번에 비해 ACCURACY가 전체적으로는 높지만, 특정 부류는 낮기도 한다.
# 성능이 약간이나마 좋아졌다.
```

```
Accuracy of plane : 39 %
Accuracy of   car : 80 %
Accuracy of  bird : 35 %
Accuracy of   cat : 39 %
Accuracy of  deer : 42 %
Accuracy of   dog : 62 %
Accuracy of  frog : 64 %
Accuracy of horse : 49 %
Accuracy of  ship : 70 %
Accuracy of truck : 62 %
```

```
#3-3
# 데이터를 변형시킬 때 사용, list를 tensor로 만들어주고 여기에 정규화 진행.
# Compose 는 여러 transform 들을 chaining, 즉 여러 transform을 진행.

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
# 변형데이터 추가, 이때 colorJitter와  랜덤하게 HorizontalFilp을 적용
transform_large = transforms.Compose(
    [transforms.ColorJitter(hue=.05, saturation=.05),transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

# CIFAR10 dataset에서 data를 다운 받아옴 이는 훈련집합
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainset_large = torchvision.datasets.CIFAR10(root = './data', train = True,
                                              download = True, transform = transform_large)
# 기존 데이터와 변형데이터를 합쳐서 데이터 샘플 개수를 늘림.
trainset_all = torch.utils.data.ConcatDataset([trainset, trainset_large])

# 훈련집합에서 4개의 데이터를 불러온다. 이때 데이터를 섞어서 불러온다, 2개의 하위 프로세스 사용
trainloader = torch.utils.data.DataLoader(trainset_all, batch_size=4,
                                          shuffle=True, num_workers=2)
# 검증집합을 정의 _CIFAR10에서 데이터를 받아옴.
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
# 검증집합에서 4개의 데이터를 불러온다, 2개의 하위 프로세스 사용
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')
```

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

3-3에서 모델은 2번 모델과 동일함으로 생략합니다.

```python
# 3-3(3)
import torch.optim as optim

# loss함수로 croosentropy사용
criterion = nn.CrossEntropyLoss()
# SGD 적용, 학습률= 0.001, 모멘텀은 0.9(수렴 속도 향상 효과)
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum = 0.9)
for epoch in range(2):
    # 학습이 진행되는지 알기위헤 0으로 초기화
    running_loss = 0.0
    for i, data in enumerate(trainloader,0):
        inputs,labels = data
        # gradient 초기화
        optimizer.zero_grad()
        # 정의한 모델로 학습
        outputs = net(inputs)
        # 손실값
        loss = criterion(outputs, labels)
        # backward 진행
        loss.backward()
        # optimiezer 진행
        optimizer.step()
```

```python
        running_loss += loss.item()
        # 천개 단위로 손실이 어떻게 나타나는 지 출력, 즉 12000개 샘플에 관한 표시
        if i % 1000 == 999:
            print("[%d, %5d] loss: %.3f" % (epoch+1,i+1,running_loss/1000))
            running_loss = 0.0
print('Finished Training')
# 2번에서 1.296 였던 loss가 데이터 확대로 인해 1.170으로 줄어든 모습을 볼 수있다
# 이는 기존의 12000 -> 25000의 데이터 확대로 인한 모습이다.
# 성능이 향상되었다.
```

```
[1,  1000] loss: 2.197
[1,  2000] loss: 1.949
[1,  3000] loss: 1.814
[1,  4000] loss: 1.725
[1,  5000] loss: 1.702
[1,  6000] loss: 1.628
[1,  7000] loss: 1.644
[1,  8000] loss: 1.579
[1,  9000] loss: 1.569
[1, 10000] loss: 1.556
[1, 11000] loss: 1.528
[1, 12000] loss: 1.541
[1, 13000] loss: 1.493
[1, 14000] loss: 1.473
[1, 15000] loss: 1.456
[1, 16000] loss: 1.485
[1, 17000] loss: 1.409
[1, 18000] loss: 1.435
[1, 19000] loss: 1.412
[1, 20000] loss: 1.360
[1, 21000] loss: 1.387
[1, 22000] loss: 1.363
[1, 23000] loss: 1.382
[1, 24000] loss: 1.351
[1, 25000] loss: 1.325
```

```
[2,  1000] loss: 1.317
[2,  2000] loss: 1.309
[2,  3000] loss: 1.316
[2,  4000] loss: 1.277
[2,  5000] loss: 1.273
[2,  6000] loss: 1.246
[2,  7000] loss: 1.278
[2,  8000] loss: 1.261
[2,  9000] loss: 1.250
[2, 10000] loss: 1.257
[2, 11000] loss: 1.233
[2, 12000] loss: 1.222
[2, 13000] loss: 1.235
[2, 14000] loss: 1.176
[2, 15000] loss: 1.201
[2, 16000] loss: 1.177
[2, 17000] loss: 1.180
[2, 18000] loss: 1.202
[2, 19000] loss: 1.186
[2, 20000] loss: 1.190
[2, 21000] loss: 1.184
[2, 22000] loss: 1.192
[2, 23000] loss: 1.159
[2, 24000] loss: 1.186
[2, 25000] loss: 1.170
Finished Training
```

```python
# 3-3(6)
# 검증 시작
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        # data는 이미지와 라벨로 구성
        images, labels = data
        # images를 학습
        outputs = net(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs.data,1)
        total += labels.size(0)
        # 정답 개수 확인
        correct += (predicted == labels).sum().item()
# 정답률 출력
print('Accuracy of the network on the 10000 test images: %d %%' % (100*correct/total))
# 2번에서는 52 % 였던 ACCURACY가 59% 로 상승했다
# 성능이 좋아졌다. 유의미한 결과의 차이가 나타났다.
```

```
Accuracy of the network on the 10000 test images: 59 %
```

```python
# 3-3(7)
# 검증 시작
# 이번에는 10개의 라벨에 관한 정답률 확인
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:
         # data는 이미지와 라벨로 구성
        images, labels = data
        # image 학습
        outputs = net(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs, 1)
        # squeeze() = 차원의 사이즈가 '1'인 차원을 제거. 여기서는 squeeze 를 호출하지 않아도 동일한 결과가 나옴.
        c = (predicted == labels).squeeze()
        # batchsize가 4이기 때문에.
        for i in range(4):
            label = labels[i]
            # label 과 predict가 일치한 경우를 더함
            class_correct[label] += c[i].item()
            # 한 라벨에 몇개를 검증 했는지에 대한 값
            class_total[label] += 1
# 결과
for i in range(10):
    print('Accuracy of %5s : %2d %%' %(classes[i], 100 * class_correct[i] / class_total[i]))
# 여전히 특정 부류는 비교적 우수한 성능을 보이지만 특정부류 문제에서는 성능이 낮다. (이는 2번과 동일하다.)
# 전반적으로 각 부류의 분류가 이전보다는 비교적 잘된다. 하지만 특정 동물들은 여전히 ACCURACY가 낮다
# 성능이 개선되었다.
```

```
Accuracy of plane : 57 %
Accuracy of   car : 74 %
Accuracy of  bird : 43 %
Accuracy of   cat : 27 %
Accuracy of  deer : 59 %
Accuracy of   dog : 50 %
Accuracy of  frog : 78 %
Accuracy of horse : 59 %
Accuracy of  ship : 75 %
Accuracy of truck : 71 %
```

```
# 3-4 CONV층에 배치정규화 수행
# CNN모델을 정의
class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()
        # 3 input image channel, 6 output channels, 5x5 square convolution 커널사이즈
        self.conv1 = nn.Conv2d(3, 6, 5)
        # pooling은 2x2
        self.pool = nn.MaxPool2d(2, 2)
        # 6 input image channel, 16 output channels, 5x5 square convolution 커널사이즈
        self.conv2 = nn.Conv2d(6, 16, 5)
        # y = Wx + b, 연산을 수행 3번의 수행을 거쳐서 마지막  특징값의 크기는 10  Linear을 연산을 3번 수행함으로
        # 차원의 축소, 즉 최종적으로 10크기를 얻음.
        self.bn1 = nn.BatchNorm2d(6)
        self.bn2 = 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):
        # conv1구조로 pooling 이때 relu 함수를 적용
        x = self.conv1(x)
        # RELU 함수를 적용하기전 배치정규화 실행, 더 효과적이기 때문이다.
        x = self.pool(F.relu(self.bn1(x)))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        # row의 수는 col 안다면 자동으로 계산. col 의 수눈 num_flat_featrues를 통해 구함. 16*5*5를 얻음
        # 즉 연산을 하기위해 데이터의 차원을 조정
        x = x.view(-1, 16 * 5 * 5)
        # y = Wx + b한 값을 relu함수로 적용
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        # 10 크기의 값이 구해진다.
        x = self.fc3(x)
        return x
# 2번 모델과 다른점은 매 CONV층 마다 배치정규화를 실행 하였다는 것이다.
net = Net2()
```

```
# 3-4(3)
import torch.optim as optim

# loss함수로 croosentropy사용
criterion = nn.CrossEntropyLoss()
# SGD 적용, 학습률= 0.001, 모멘텀은 0.9(수렴 속도 향상 효과)
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum = 0.9)
for epoch in range(2):
    # 학습이 진행되는지 알기위헤 0으로 초기화
    running_loss = 0.0
    for i, data in enumerate(trainloader,0):
        inputs,labels = data
        # gradient 초기화
        optimizer.zero_grad()
        # 정의한 모델로 학습
        outputs = net(inputs)
        # 손실값
        loss = criterion(outputs, labels)
        # backward 진행
        loss.backward()
        # optimiezer 진행
        optimizer.step()
```

```
        running_loss += loss.item()
        # 천개 단위로 손실이 어떻게 나타나는 지 출력, 즉 12000개 샘플에 관한 표시
        if i % 1000 == 999:
            print("[%d, %5d] loss: %.3f" % (epoch+1,i+1,running_loss/1000))
            running_loss = 0.0
print('Finished Training')
# 2번에서 최종 값은 1.296 이었다. 여기서는 1.358을 얻은 것으로 보았을 때 성능이 다소 떨어짐이 보인다
# 다만 그 차이는 크지는 않고, 이외에는 비슷한 loss감소 속도를 보인다. 그리고 학습이 적절하게 이루어 졌다.
# 하지만 속도는 더빨라진다.
# loss가 2번에 비해서 더 크기 때문에 2번에 비해 ACCURACY도 더 작게 나온다.
```

```
[1,  1000] loss: 2.245
[1,  2000] loss: 1.978
[1,  3000] loss: 1.847
[1,  4000] loss: 1.758
[1,  5000] loss: 1.715
[1,  6000] loss: 1.678
[1,  7000] loss: 1.651
[1,  8000] loss: 1.640
[1,  9000] loss: 1.604
[1, 10000] loss: 1.593
[1, 11000] loss: 1.547
[1, 12000] loss: 1.542
[2,  1000] loss: 1.495
[2,  2000] loss: 1.481
[2,  3000] loss: 1.499
[2,  4000] loss: 1.458
[2,  5000] loss: 1.429
[2,  6000] loss: 1.424
[2,  7000] loss: 1.438
[2,  8000] loss: 1.391
[2,  9000] loss: 1.393
[2, 10000] loss: 1.401
[2, 11000] loss: 1.386
[2, 12000] loss: 1.358
Finished Training
```

```
# 3-4(6)
# 검증 시작
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        # data는 이미지와 라벨로 구성
        images, labels = data
        # images를 학습
        outputs = net(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs.data,1)
        total += labels.size(0)
        # 정답 개수 확인
        correct += (predicted == labels).sum().item()
# 정답률 출력
print('Accuracy of the network on the 10000 test images: %d %%' % (100*correct/total))
# 2번에서는 52 % 였던 ACCURACY가 더 낮아졌다
# 성능이 하락이 되었다. 하지만 크게 차이가 나지는 않는다.
```

```
Accuracy of the network on the 10000 test images: 51 %
```

```
# 3-4(7)
# 검증 시작
# 이번에는 10개의 라벨에 관한 정답를 확인
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:
        # data는 이미지와 라벨로 구성
        images, labels = data
        # image 학습
        outputs = net(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs, 1)
        # squeeze() = 차원의 사이즈가 '1'인 차원을 제거. 여기서는 squeeze 를 호출하지 않아도 동일한 결과가 나옴.
        c = (predicted == labels).squeeze()
        # batchsize가 4이기 때문에.
        for i in range(4):
            label = labels[i]
            # label 과 predict가 일치한 경우를 더함
            class_correct[label] += c[i].item()
            # 한 라벨에 몇개를 검증 했는지에 대한 값
            class_total[label] += 1
```

```
# 결과
for i in range(10):
    print('Accuracy of %5s : %2d %%' %(classes[i], 100 * class_correct[i] / class_total[i]))
# 특정 부류는 비교적 우수한 성능을 보이지만 특정부류 문제에서는 성능이 낮다. (이는 2번과 동일하다.)
# 전체적으로 따진다면 2번보다 ACCURACY가 낮다.
# 따라서 성능이 좋지 않다.
# 학습률을 더크게 한다던지의 변화를 준다면 좀더 높음 ACCURACY를 얻을 수 있다.
```

```
Accuracy of plane : 60 %
Accuracy of   car : 76 %
Accuracy of  bird : 29 %
Accuracy of   cat : 16 %
Accuracy of  deer : 44 %
Accuracy of   dog : 60 %
Accuracy of  frog : 53 %
Accuracy of horse : 60 %
Accuracy of  ship : 59 %
Accuracy of truck : 52 %
```

3-5는 2번과 모델이 동일함으로 생략합니다.

```
# 3-5(3) 로그우도 적용
import torch.optim as optim
# NLLLoss를 이용할 때는 logSoftmax를 이용하여 구현해야 한다는 것이다.
# 이는 LOSS가 음수가 아닌 양수에 수렴하게끔 하기 위해서이다.
criterion = nn.NLLLoss()
m = nn.LogSoftmax()
# SGD 적용, 학습률= 0.001, 모멘텀은 0.9(수렴 속도 향상 효과)
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum = 0.9)
for epoch in range(2):
    # 학습이 진행되는지 알기위헤 0으로 초기화
    running_loss = 0.0
    for i, data in enumerate(trainloader,0):
        inputs,labels = data
        # gradient 초기화
        optimizer.zero_grad()
        # 정의한 모델로 학습
        outputs = net(inputs)
        # 손실값
        loss = criterion(m(outputs),labels)
        # backward 진행
        loss.backward()
        # optimiezer 진행
        optimizer.step()
```

```
        running_loss += loss.item()
        # 천개 단위로 손실이 어떻게 나타나는 지 출력, 즉 12000개 샘플에 관한 표시
        if i % 1000 == 999:
            print("[%d, %5d] loss: %.3f" % (epoch+1,i+1,running_loss/1000))
            running_loss = 0.0
print('Finished Training')
# 2번과 비교했을 때 초기에, loss가 줄어드는 폭이 작으나, 마지막에는 약간의 나은성능 loss = 1.234를 얻는다
# 성능이 크게 나아졌다고 평가하기는 부족하고, 비슷한 성능을 가졌다고 할 수 있다.
# 학습은 loss가 줄어들기 때문에 적절하다.
```

```
[1,  1000] loss: 2.288
[1,  2000] loss: 2.105
[1,  3000] loss: 1.989
[1,  4000] loss: 1.903
[1,  5000] loss: 1.777
[1,  6000] loss: 1.695
[1,  7000] loss: 1.619
[1,  8000] loss: 1.555
[1,  9000] loss: 1.507
[1, 10000] loss: 1.481
[1, 11000] loss: 1.444
[1, 12000] loss: 1.431
[2,  1000] loss: 1.376
[2,  2000] loss: 1.336
[2,  3000] loss: 1.362
[2,  4000] loss: 1.324
[2,  5000] loss: 1.315
[2,  6000] loss: 1.318
[2,  7000] loss: 1.285
[2,  8000] loss: 1.259
[2,  9000] loss: 1.308
[2, 10000] loss: 1.274
[2, 11000] loss: 1.251
[2, 12000] loss: 1.234
Finished Training
```

```python
# 3-5(6)
# 검증 시작
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        # data는 이미지와 라벨로 구성
        images, labels = data
        # images를 학습
        outputs = net(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs.data,1)
        total += labels.size(0)
        # 정답 개수 확인
        correct += (predicted == labels).sum().item()
# 정답률 출력
print('Accuracy of the network on the 10000 test images: %d %%' % (100*correct/total))
# loss의 양상이 비슷함으로 accuracy 도 비슷하다. 다만 여기서는 56%로 기존의 52%에 비해 accuracy가 소폭 상승했다.
```

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

```python
# 3-5(7)
# 이번에는 10개의 라벨에 관한 정답률 확인
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:
         # data는 이미지와 라벨로 구성
        images, labels = data
        # image 학습
        outputs = net(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs, 1)
        # squeeze() = 차원의 사이즈가 '1'인 차원을 제거. 여기서는 squeeze 를 호출하지 않아도 동일한 결과가 나옴.
        c = (predicted == labels).squeeze()
        # batchsize가 4이기 때문에.
        for i in range(4):
            label = labels[i]
            # label 과 predict가 일치한 경우를 더함
            class_correct[label] += c[i].item()
            # 한 라벨에 몇개를 검증 했는지에 대한 값
            class_total[label] += 1
```

```python
# 결과
for i in range(10):
    print('Accuracy of %5s : %2d %%' %(classes[i], 100 * class_correct[i] / class_total[i]))
# 특정 부류는 비교적 우수한 성능을 보이지만 특정부류 문제에서는 성능이 낮다. (이는 2번과 동일하다.)
# ACUURACY가 소폭 상승했다.
# 따라서 성능이 좋지 않다. 여전히 비슷한 양상을 보인다.
```

```
Accuracy of plane : 59 %
Accuracy of   car : 67 %
Accuracy of  bird : 25 %
Accuracy of   cat : 46 %
Accuracy of  deer : 51 %
Accuracy of   dog : 42 %
Accuracy of  frog : 67 %
Accuracy of horse : 67 %
Accuracy of  ship : 79 %
Accuracy of truck : 59 %
```

3-6 의 모델은 2번과 동일함으로 생략합니다.

```python
# 3-6(3) L2놈 적용
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
# SGD 적용, 학습률= 0.001, 모멘텀은 0.9(수렴 속도 향상 효과)
# l2norm 은 weight_decay 값을 주어서 해결할 수 있습니다. 이는 수학적으로 증명이 되어있습니다.
# 즉 학습 파라메터의 값이 크면 그에 상응하는 큰 패널티를 부여하는 것입니다.
# 이때 대표적으로 0.001, 0.01, 0.1, 1, and 10. 과 같은 값들을 주어서, 비교를 통해 최적값을 구할 수 있습니다.
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum = 0.9,weight_decay=1e-4)
for epoch in range(2):
    # 학습이 진행되는지 알기위해 0으로 초기화
    running_loss = 0.0
    for i, data in enumerate(trainloader,0):
        inputs,labels = data
        # gradient 초기화
        optimizer.zero_grad()
        # 정의한 모델로 학습
        outputs = net(inputs)
        # 손실값
        loss = criterion(outputs,labels)
        # backward 진행
        loss.backward()
        # optimiezer 진행
        optimizer.step()
```

```python
        running_loss += loss.item()
        # 천개 단위로 손실이 어떻게 나타나는 지 출력, 즉 12000개 샘플에 관한 표시
        if i % 1000 == 999:
            print("[%d, %5d] loss: %.3f" % (epoch+1,i+1,running_loss/1000))
            running_loss = 0.0
print('Finished Training')
# 2번에 비해 초기 LOSS 감소폭이 작은 경향이 있다. 하지만 여전히 최종적인 결과는 비슷하다.
# 여기선 1.313이 나온다. 이는 2번의 최종값 1.296보다는 크다.
# 하지만 이는 decay 값의 변경에 따라 더 최적의 값이 존재 할 수 있다
# 학습은 적잘하게 이루어지고 있다.
```

```
[1,  1000] loss: 2.302
[1,  2000] loss: 2.204
[1,  3000] loss: 1.959
[1,  4000] loss: 1.822
[1,  5000] loss: 1.733
[1,  6000] loss: 1.674
[1,  7000] loss: 1.610
[1,  8000] loss: 1.590
[1,  9000] loss: 1.580
[1, 10000] loss: 1.527
[1, 11000] loss: 1.489
[1, 12000] loss: 1.499
[2,  1000] loss: 1.459
[2,  2000] loss: 1.394
[2,  3000] loss: 1.406
[2,  4000] loss: 1.394
[2,  5000] loss: 1.363
[2,  6000] loss: 1.378
[2,  7000] loss: 1.350
[2,  8000] loss: 1.324
[2,  9000] loss: 1.319
[2, 10000] loss: 1.337
[2, 11000] loss: 1.307
[2, 12000] loss: 1.313
Finished Training
```

```python
# 3-6(6)
# 검증 시작
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        # data는 이미지와 라벨로 구성
        images, labels = data
        # images를 학습
        outputs = net(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs.data,1)
        total += labels.size(0)
        # 정답 개수 확인
        correct += (predicted == labels).sum().item()
# 정답률 출력
print('Accuracy of the network on the 10000 test images: %d %%' % (100*correct/total))
# Accuracy 는 54%로 기존의 52 % 에 비해 소폭 상승 했다.
# 하지만 이는 크게 성능의 차이가 난다고 하기는 힘들다. 샘플에 따라, accuracy가 달라지기 때문이다.
# 따라서 큰 성능의 향상은 없었다.
# 다만 여기서도 decay값을 변경하여 적용한다면 성능 향상을 기대 할 수도 있다.
```

```
Accuracy of the network on the 10000 test images: 54 %
```

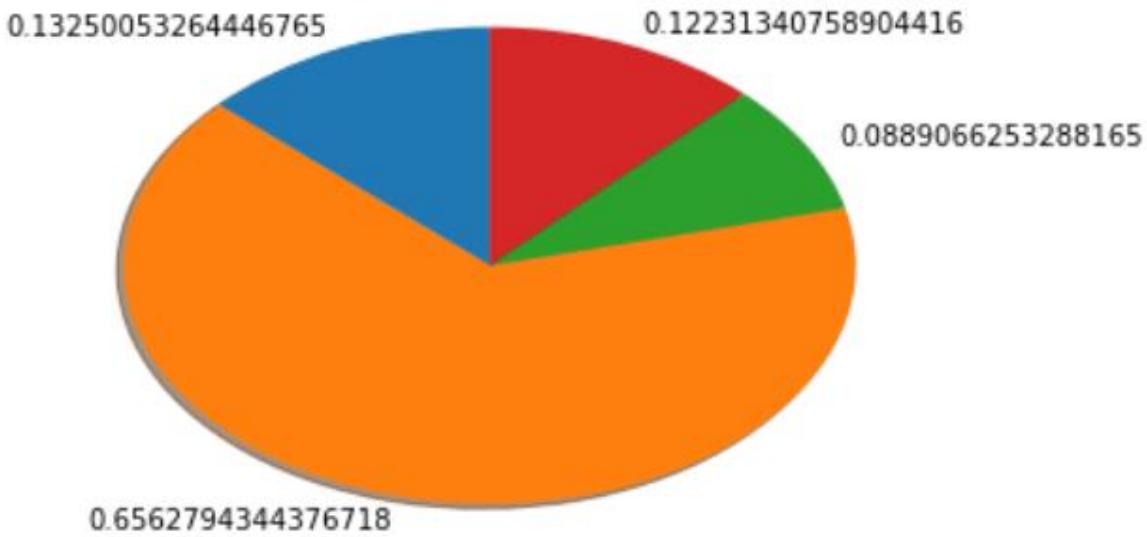
```python
# 3-6(7)
# 이번에는 10개의 라벨에 관한 정답률 확인
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        # data는 이미지와 라벨로 구성
        images, labels = data
        # image 학습
        outputs = net(images)
        # 예측 결과, 결과벡터에서 1행마다 가장 큰 값을 출력. 이때 predicted는 그 때의 행에서 열의 위치.
        _, predicted = torch.max(outputs, 1)
        # squeeze() = 차원의 사이즈가 '1'인 차원을 제거. 여기서는 squeeze 를 호출하지 않아도 동일한 결과가 나옴.
        c = (predicted == labels).squeeze()
        # batchsize가 4이기 때문에.
        for i in range(4):
            label = labels[i]
            # label 과 predict가 일치한 경우를 더함
            class_correct[label] += c[i].item()
            # 한 라벨에 몇개를 검증 했는지에 대한 값
            class_total[label] += 1
```

```python
# 결과
for i in range(10):
    print('Accuracy of %5s : %2d %%' %(classes[i], 100 * class_correct[i] / class_total[i]))
# 특정 부류는 비교적 우수한 성능을 보이지만 특정부류 문제에서는 성능이 낮다. (이는 2번과 동일하다.)
# 다만 2번에서는 극단적으로 낮았던 부류의 일치율이, 다소 개선되어 있다.
# 즉 ACCURACY가 소폭 상승했다.
# 하지만 여전히 성능이 좋지는 않다. 여전히 비슷한 양상을 보인다.
```

```
Accuracy of plane : 62 %
Accuracy of   car : 75 %
Accuracy of  bird : 53 %
Accuracy of   cat : 35 %
Accuracy of  deer : 39 %
Accuracy of   dog : 31 %
Accuracy of  frog : 67 %
Accuracy of horse : 69 %
Accuracy of  ship : 71 %
Accuracy of truck : 44 %
```

```python
# 4 소프트맥스 계산
import numpy as np
import matplotlib.pyplot as plt

def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum()

x = np.array([0.4,2.0,0.001,0.32])

y = softmax(x)

print(np.sum(y))

ratio = y
labels = y

plt.pie(ratio, labels=labels, shadow=True, startangle=90)
plt.show()
```

```
1.0
```



```python
# 5-1 MSELoss
import math

x =[0,0,0,1]
y = [0.001, 0.9,0.001,0.098]
sum = 0
for a,b in zip(x,y):
    sum += ((a-b)**2)
mse = sum/2
print(mse)
# 강의노트 5-1의 공식 활용
```

```
0.811803
```

```python
# 5-2 CrossEntropyLoss
out =[0,0,0,1]
y = [0.001, 0.9,0.001,0.098]
sum = 0
for b,a in zip(out,y):
    sum -= (b * math.log(a, 2)+(1-b) * math.log(1-a, 2))
print(sum)
```

```
6.675889369173579
```

```python
# 5-3 로그우도
# 5.8의 설명에 따라서 oy라는 하나의 노드만 적용하면 된다. 따라서 다음과 같이 진행한다.
# oy 는 샘플의 레이블에 해당하는 실제 값, 이에 대응되는 것은 4번째 값인 0.098 이다
# oy 에 대하여 -log2 (oy) 를 출력하면 다음과 같다.
print('로그우도 = ', - math.log(0.098, 2))
```

```
로그우도 =  3.3510744405468786
```