

# 기계학습개론 Final Project 보고서

IT 공학전공 2216899 정지윤

## 목차

- I. Network Model (Batch Size, Epoch, Dropout)
- II. Data Processing (Normalization, Augmentation, Initialization)
- III. Optimizer/Scheduler
- IV. Final Code Description

CIFAR10 을 학습할 때 팀원들과 함께 정했던 주요 고려사항으로는 크게 네 부분이 있는데, Network Model, Data Processing, Epoch, Optimizer/Scheduler 로 나눌 수 있다. 매번 나오는 결과에 따라 팀원들끼리 서로 다른 부분을 맡아서 실습했기 때문에 모든 실습이 아래의 순서대로 진행되지는 않았지만, 최종 결과가 나오기까지 그동안의 실습 과정을 카테고리 별로 나눠서 설명하는 것이 좋을 것 같다고 판단하여 네 부분으로 나눠보았다.

### I. Network Model (Batch Size, Epoch, Dropout)

먼저 모델 선정에 관한 부분이다. 팀원들과 함께 테스트해봤던 모델로는 VGG, ResNet, DenseNet 등이 있다. 다른 모델들도 시도해보긴 하였지만 CIFAR10 은 깊은 구조를 가진 모델이 적합할 것이라고 판단하여 세 가지로 추려서 탐구해보기로 결정하였다. 또한, 모델에 따라 batch size 와 epoch 을 몇으로 설정해야 GPU 용량이나 시간이 너무 많이 들지 않으면서 동시에 성능이 좋게 나오는지, dropout 을 적용하는 것이 좋을지도 함께 고려해보았다.

#### ① VGG11

```
11 transforms_cifar10 = transforms.Compose([transforms.Resize((32, 32)),
12                                     transforms.ToTensor(),
13                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
14                                     ])

# Train dataset
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transforms_cifar10)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, num_workers=2)

# Test dataset
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transforms_cifar10)
testloader = torch.utils.data.DataLoader(testset, batch_size=128, shuffle=False, num_workers=2)

# Classes of CIFAR-10 dataset
classes = ("plane", "car", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck")

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170408071/170408071 [00:05<00:00] 28747552.04it/s]
```

✓  
0초

```
# Loss function and optimizer
loss_fun = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.0001)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
```

## ▼ Train the network

Train your own network using the above loss function and optimizer.

✓  
5분

```
[13] # Train the model
epochs = 20 # number of epochs

for epoch in range(epochs):

    loss_tmp = 0.0
    epoch_loss = 0.0
```

✓  
5분

```
# Update the learning rate according to the learning rate scheduler
scheduler.step()

# Print the epoch loss
print('[Epoch - %d] Loss: %.3f' %(epoch + 1, epoch_loss / (i+1)))

print('Finished Training')
```

```
[Epoch - 1] Loss: 0.043
[Epoch - 2] Loss: 0.034
[Epoch - 3] Loss: 0.033
[Epoch - 4] Loss: 0.032
[Epoch - 5] Loss: 0.032
[Epoch - 6] Loss: 0.029
[Epoch - 7] Loss: 0.028
[Epoch - 8] Loss: 0.030
[Epoch - 9] Loss: 0.028
[Epoch - 10] Loss: 0.026
[Epoch - 11] Loss: 0.007
[Epoch - 12] Loss: 0.001
[Epoch - 13] Loss: 0.001
[Epoch - 14] Loss: 0.001
[Epoch - 15] Loss: 0.000
[Epoch - 16] Loss: 0.000
[Epoch - 17] Loss: 0.000
```

▶

```
# Test the trained model with overall test dataset

correct = 0
total = 0
for data in testloader:
    # Load the data
    inputs_test, labels_test = data
    inputs_test = inputs_test.to(device)
    labels_test = labels_test.to(device)

    # Estimate the output using the trained network
    outputs_test = net(inputs_test)
    _, predicted = torch.max(outputs_test.data, 1)

    # Calculate the accuracy
    total += labels_test.size(0)
    correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

## [SimpleNet / Training 5 epochs] Accuracy of the network on the 10,000 test images: 9 %
## [VGG11 / Training 5 epochs] Accuracy of the network on the 10,000 test images: 11 %
```

▶

Accuracy of the network on the 10,000 test images: 79 %

최초에 학습했던 모델은 VGG11 이었다. 이때는 배치사이즈를 128, 에폭을 20, lr=0.0001 로 수정하였는데 정확도가 79%가 나왔다.

그래서 배치를 64 로, lr=0.001 로 수정해보았더니 80%로 오르는 것을 확인할 수 있었다. 여러 번 비교해본 결과 lr 은 0.001 이 가장 적당하고, 배치사이즈는 너무 크지 않은 64~128 정도의 값이 적당한 것 같다는 결론이 나왔다.

The screenshot displays a Jupyter Notebook with the following content:

```

transforms_cifar10 = transforms.Compose([transforms.Resize((32, 32)),
                                         transforms.ToTensor(),
                                         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                                         ])

# Train dataset
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transforms_cifar10)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True, num_workers=2)

# Test dataset
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transforms_cifar10)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False, num_workers=2)

# Classes of CIFAR-10 dataset
classes = ("plane", "car", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck")

Files already downloaded and verified
Files already downloaded and verified

[46] # Examples of dataset

def imshow(img):
    img = img / 2 + 0.5

# Update the learning rate according to the learning rate scheduler
scheduler.step()

# Print the epoch loss
print('Epoch - %d Loss: %.3f' % (epoch + 1, epoch_loss / (i+1)))

print('Finished Training')

Epoch - 1] Loss: 1.136
Epoch - 2] Loss: 0.726
Epoch - 3] Loss: 0.516
Epoch - 4] Loss: 0.360
Epoch - 5] Loss: 0.240
Epoch - 6] Loss: 0.163
Epoch - 7] Loss: 0.121
Epoch - 8] Loss: 0.102
Epoch - 9] Loss: 0.088
Epoch - 10] Loss: 0.075
Epoch - 11] Loss: 0.021
Epoch - 12] Loss: 0.005
Epoch - 13] Loss: 0.003
Epoch - 14] Loss: 0.002
Epoch - 15] Loss: 0.001
Epoch - 16] Loss: 0.002
Epoch - 17] Loss: 0.001
Epoch - 18] Loss: 0.001
Epoch - 19] Loss: 0.000
Epoch - 20] Loss: 0.000
Finished Training

- Test the network

correct = 0
total = 0
for data in testloader:
    # Load the data
    inputs_test, labels_test = data
    inputs_test = inputs_test.to(device)
    labels_test = labels_test.to(device)

    # Estimate the output using the trained network
    outputs_test = net(inputs_test)
    _, predicted = torch.max(outputs_test.data, 1)

    # Calculate the accuracy
    total += labels_test.size(0)
    correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

## [SimpleNet / Training 5 epochs] Accuracy of the network on the 10,000 test images: 9 %
## [VGG11 / Training 5 epochs] Accuracy of the network on the 10,000 test images: 11 %

Accuracy of the network on the 10,000 test images: 80 %
  
```

## ② VGG19

VGG19 모델은 처음에 해봤을 때 에폭 20~30, 배치사이즈 64 와 256 이 성능이 80 대로 안정적으로 나왔었다. 그리고 확실히 vgg 는 다른 모델에 비해 시간이 적게 걸린다는 것을 알 수 있었다.

```
transforms_cifar10 = transforms.Compose([transforms.Resize((32, 32)),
                                       transforms.ToTensor(),
                                       transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                                       ])

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Train dataset
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transforms_cifar10)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True, num_workers=2)

# Test dataset
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transforms_cifar10)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False, num_workers=2)

# Classes of CIFAR-10 dataset
classes = ("plane", "car", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck")

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:02<00:00, 71737689.15it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified

[4] # Examples of dataset

def imshow(img):
```

```
# Test the trained model with overall test dataset

correct = 0
total = 0
for data in testloader:
    # Load the data
    inputs_test, labels_test = data
    inputs_test = inputs_test.to(device)
    labels_test = labels_test.to(device)

    # Estimate the output using the trained network
    outputs_test = net(inputs_test)
    _, predicted = torch.max(outputs_test.data, 1)

    # Calculate the accuracy
    total += labels_test.size(0)
    correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

## [SimpleNet / Training 5 epochs] Accuracy of the network on the 10,000 test images: 9 %
## [VGG11 / Training 5 epochs] Accuracy of the network on the 10,000 test images: 11 %

Accuracy of the network on the 10,000 test images: 85 %
```

VGG19 배치 64 에폭 20 85%

```
transforms_cifar10 = transforms.Compose([transforms.Resize((32, 32)), #channel 32x32
                                       transforms.ToTensor(),
                                       transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                                       ])

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.5, contrast=0.3, saturation=0.2, hue=0.3),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Train dataset
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transforms_cifar10)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=256, shuffle=True, num_workers=2)

# Test dataset
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transforms_cifar10)
testloader = torch.utils.data.DataLoader(testset, batch_size=256, shuffle=False, num_workers=2)

# Classes of CIFAR-10 dataset
classes = ("plane", "car", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck")

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:01<00:00, 95066703.78it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified

[4] # Examples of dataset
```

```
cfg = {'VGG19': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 512, 512, 'M']}

class VGG19(nn.Module):
    def __init__(self):
        super(VGG19, self).__init__()
        self.features = self.make_layers(cfg['VGG19'])
        self.classifier = nn.Linear(512, 10)

    def make_layers(self, cfg):
        layers = []
        in_channels = 3
        for x in cfg:
            if x == 'M':
                layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
            else:
                layers += [nn.Conv2d(in_channels, x, kernel_size=3, padding=1),
                           nn.BatchNorm2d(x),
                           nn.ReLU(inplace=True)]
                in_channels = x
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.features(x)
        out = out.view(out.size(0), -1)
        out = self.classifier(out)
        return out

net = VGG19().to(device)

# Loss function and optimizer
loss_fun = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.001)

#stepLR
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

#lambdaLR
#scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer=optimizer, lr_lambda=lambda epoch: 0.95 ** epoch)

#MultiplicativeLR
#scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer=optimizer, lr_lambda=lambda epoch: 0.95 ** epoch)

#MultiStepLR
#scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[30, 80], gamma=0.5)

# Train the network
Train your own network using the above loss function and optimizer.

[18] # Train the model
epochs = 30 # number of epochs

for epoch in range(epochs):
    loss_tmp = 0.0

# Test the trained model with overall test dataset

correct = 0
total = 0
for data in testloader:
    # Load the data
    inputs_test, labels_test = data
    inputs_test = inputs_test.to(device)
    labels_test = labels_test.to(device)

    # Estimate the output using the trained network
    outputs_test = net(inputs_test)
    _, predicted = torch.max(outputs_test.data, 1)

    # Calculate the accuracy
    total += labels_test.size(0)
    correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

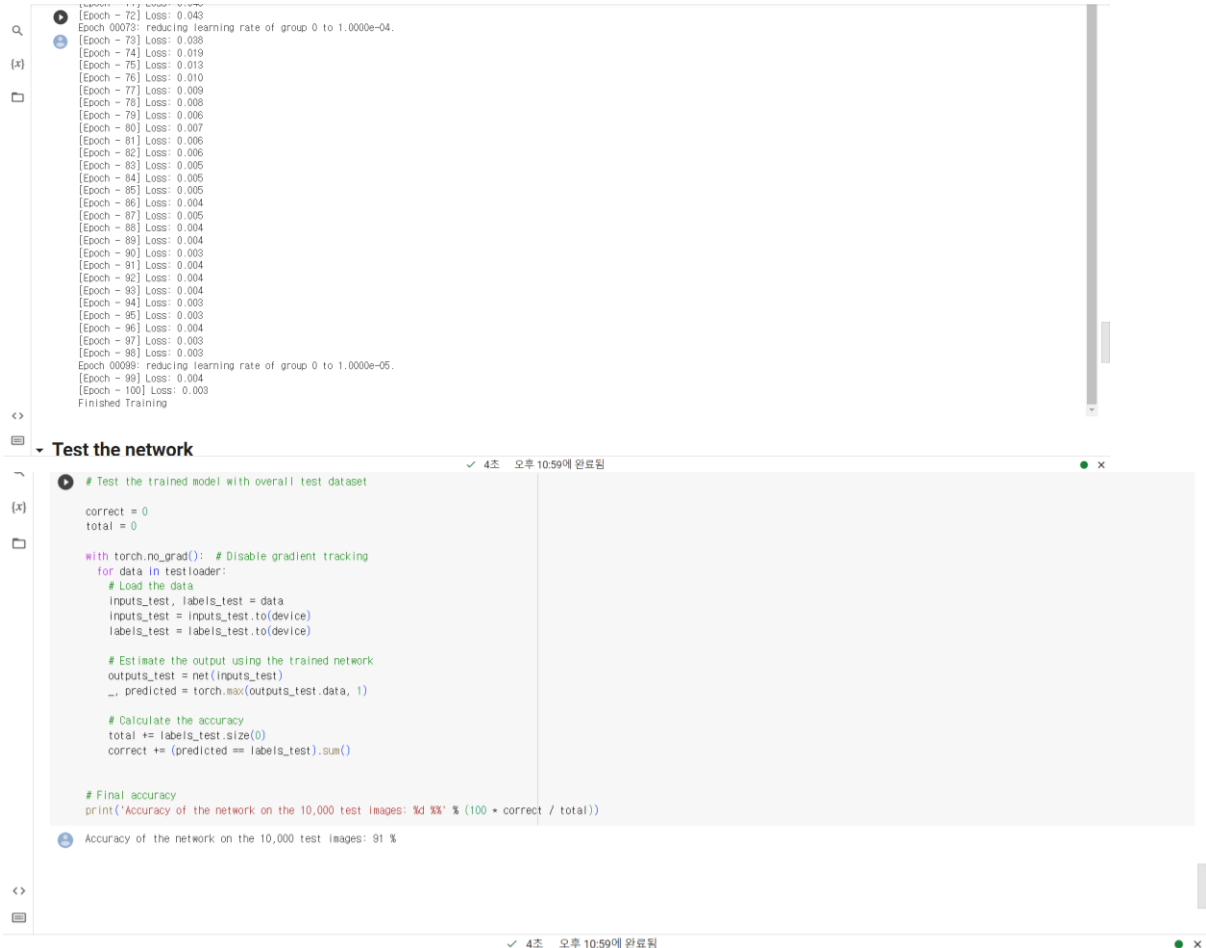
## [SimpleNet / Training 5 epochs] Accuracy of the network on the 10,000 test images: 9 %
## [VGG11 / Training 5 epochs] Accuracy of the network on the 10,000 test images: 11 %

Accuracy of the network on the 10,000 test images: 87 %
```

VGG19 배치 256 에폭 30 87%

그리고 초반에 드롭아웃 적용을 했더니 성능이 확 줄었다. 그래서 처음에는 에폭 수가 작아서 드롭아웃을 넣으면 오히려 과적합이 심해져서 정확도가 낮아지는 건가 싶었는데, 마지막에 net.eval()을 적용하지 않아서 그런 것 같다. eval() 함수를 사용하면 모델이 추론 또는 평가를 수행하는 모드로 전환되는데

이는 드롭아웃 또는 배치 정규화 같은 정규화 기법을 평가 모드에 맞게 동작시키는 역할을 한다. 하지만 eval() 함수를 사용하지 않으면 학습 모드로 유지되고 정규화 기법이 다르게 동작할 수 있다고 한다. 따라서 마지막에 net.eval()을 추가하고 에폭을 100 으로 확 늘렸더니 92%의 좋은 성능이 나왔다.



```
[Epoch - 72] Loss: 0.043
Epoch 00073: reducing learning rate of group 0 to 1.0000e-04.
[Epoch - 73] Loss: 0.038
[Epoch - 74] Loss: 0.019
[Epoch - 75] Loss: 0.013
[Epoch - 76] Loss: 0.010
[Epoch - 77] Loss: 0.009
[Epoch - 78] Loss: 0.008
[Epoch - 79] Loss: 0.006
[Epoch - 80] Loss: 0.007
[Epoch - 81] Loss: 0.006
[Epoch - 82] Loss: 0.006
[Epoch - 83] Loss: 0.005
[Epoch - 84] Loss: 0.005
[Epoch - 85] Loss: 0.005
[Epoch - 86] Loss: 0.004
[Epoch - 87] Loss: 0.005
[Epoch - 88] Loss: 0.004
[Epoch - 89] Loss: 0.004
[Epoch - 90] Loss: 0.003
[Epoch - 91] Loss: 0.004
[Epoch - 92] Loss: 0.004
[Epoch - 93] Loss: 0.004
[Epoch - 94] Loss: 0.003
[Epoch - 95] Loss: 0.003
[Epoch - 96] Loss: 0.004
[Epoch - 97] Loss: 0.003
[Epoch - 98] Loss: 0.003
Epoch 00099: reducing learning rate of group 0 to 1.0000e-05.
[Epoch - 99] Loss: 0.004
[Epoch - 100] Loss: 0.003
Finished Training
```

**Test the network**

```
# Test the trained model with overall test dataset

correct = 0
total = 0

with torch.no_grad(): # Disable gradient tracking
    for data in testloader:
        # Load the data
        inputs_test, labels_test = data
        inputs_test = inputs_test.to(device)
        labels_test = labels_test.to(device)

        # Estimate the output using the trained network
        outputs_test = net(inputs_test)
        _, predicted = torch.max(outputs_test.data, 1)

        # Calculate the accuracy
        total += labels_test.size(0)
        correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))
```

Accuracy of the network on the 10,000 test images: 91 %

net.eval 이 없을 때 91%



```
[9] # Test the trained model with overall test dataset

correct = 0
total = 0

with torch.no_grad(): # Disable gradient tracking
    net.eval()
    for data in testloader:
        # Load the data
        inputs_test, labels_test = data
        inputs_test = inputs_test.to(device)
        labels_test = labels_test.to(device)

        # Estimate the output using the trained network
        outputs_test = net(inputs_test)
        _, predicted = torch.max(outputs_test.data, 1)

        # Calculate the accuracy
        total += labels_test.size(0)
        correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))
```

Accuracy of the network on the 10,000 test images: 92 %

net.eval 이 있을 때 92%

### ③ ResNet

ResNet18,34 는 에폭 30, 배치사이즈 256 으로 진행하였는데 전체적으로 성능이 좋긴 하지만 80 후반에서 더 이상 오르지 않았다. 또한 ResNet 152 는 OutOfMemory 오류가 났었다. 아무래도 ResNet 은 GPU 용량이 적을 때 사용하기 좋은 모델은 아닌 것 같았다.



The screenshot displays a Jupyter Notebook with two cells. The top cell contains Python code for a ResNet model, and the bottom cell shows the output of the training process.

```
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)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out
```

The bottom cell shows the output of the training process, which includes a list of epochs and their corresponding loss values, followed by a "Finished Training" message.

```
[7] print('Finished Training')

[Epoch - 1] Loss: 1.344
[Epoch - 2] Loss: 0.821
[Epoch - 3] Loss: 0.605
[Epoch - 4] Loss: 0.470
[Epoch - 5] Loss: 0.366
[Epoch - 6] Loss: 0.276
[Epoch - 7] Loss: 0.205
[Epoch - 8] Loss: 0.146
[Epoch - 9] Loss: 0.114
[Epoch - 10] Loss: 0.079
[Epoch - 11] Loss: 0.023
[Epoch - 12] Loss: 0.007
[Epoch - 13] Loss: 0.004
[Epoch - 14] Loss: 0.003
[Epoch - 15] Loss: 0.003
[Epoch - 16] Loss: 0.002
[Epoch - 17] Loss: 0.002
[Epoch - 18] Loss: 0.001
[Epoch - 19] Loss: 0.001
[Epoch - 20] Loss: 0.001
[Epoch - 21] Loss: 0.001
[Epoch - 22] Loss: 0.001
[Epoch - 23] Loss: 0.001
[Epoch - 24] Loss: 0.001
[Epoch - 25] Loss: 0.001
[Epoch - 26] Loss: 0.001
[Epoch - 27] Loss: 0.001
[Epoch - 28] Loss: 0.001
[Epoch - 29] Loss: 0.001
[Epoch - 30] Loss: 0.001
Finished Training
```

```
[9] # Test the trained model with overall test dataset

correct = 0
total = 0
for data in testloader:
    # Load the data
    inputs_test, labels_test = data
    inputs_test = inputs_test.to(device)
    labels_test = labels_test.to(device)

    # Estimate the output using the trained network
    outputs_test = net(inputs_test)
    _, predicted = torch.max(outputs_test.data, 1)

    # Calculate the accuracy
    total += labels_test.size(0)
    correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

## [SimpleNet / Training 5 epochs] Accuracy of the network on the 10,000 test images: 9 %
## [VGG11 / Training 5 epochs] Accuracy of the network on the 10,000 test images: 11 %

Accuracy of the network on the 10,000 test images: 86 %
```

2초 오후 3:36에 완료됨

ResNet18 86%

```
[37]

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

def ResNet34():
    return ResNet(BasicBlock, [3, 4, 6, 3])

def ResNet50():
    return ResNet(Bottleneck, [3, 4, 6, 3])

def ResNet101():
    return ResNet(Bottleneck, [3, 4, 23, 3])

def ResNet152():
    return ResNet(Bottleneck, [3, 8, 36, 3])

def test():
    net = ResNet18()
    ... v = net(torch.randn(1, 3, 224, 224))
    total = 0
    for data in testloader:
        # Load the data
        inputs_test, labels_test = data
        inputs_test = inputs_test.to(device)
        labels_test = labels_test.to(device)

        # Estimate the output using the trained network
        outputs_test = net(inputs_test)
        _, predicted = torch.max(outputs_test.data, 1)

        # Calculate the accuracy
        total += labels_test.size(0)
        correct += (predicted == labels_test).sum()

    # Final accuracy
    print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

## [SimpleNet / Training 5 epochs] Accuracy of the network on the 10,000 test images: 9 %
## [VGG11 / Training 5 epochs] Accuracy of the network on the 10,000 test images: 11 %

Accuracy of the network on the 10,000 test images: 87 %
```

ResNet34 87%



```
def ResNet50():
    return ResNet(Bottleneck, [3, 4, 6, 3])

def ResNet101():
    return ResNet(Bottleneck, [3, 4, 23, 3])

def ResNet152():
    return ResNet(Bottleneck, [3, 8, 36, 3])

def test():
    net = ResNet101()
    y = net(torch.randn(1, 3, 32, 32))
    print(y.size())

# test()

net = ResNet152().to(device)
```

### Loss function and optimizer

Set the **loss function and optimizer** for training CNN. You can modify the loss function or optimizer for better performance.

```
[11] # Loss function and optimizer
loss_fun = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.001)

#stepLR
```

0초 오후 3:39에 완료됨

```
# Print the epoch loss
print('[Epoch - %d] Loss: %.3f' % (epoch + 1, epoch_loss / (i+1)))

print('Finished Training')
```

OutOfMemoryError Traceback (most recent call last)

```
<ipython-input-12-5610c3c5b116> in <cell line: 4>()
    14
    15     # Estimate the output using the network
--> 16     outputs = net(inputs)
    17
    18     # Calculate the loss between the output of the network and label
```

8 frames

```
~/local/lib/python3.10/dist-packages/torch/nn/functional.py in batch_norm(input, running_mean, running_var, weight, bias, training, momentum, eps)
    2448     _verify_batch_size(input.size())
    2449
-> 2450     return torch.batch_norm(
    2451         input, weight, bias, running_mean, running_var, training, momentum, eps, torch.backends.cudnn.enabled
    2452     )
```

OutOfMemoryError: CUDA out of memory. Tried to allocate 32.00 MiB (GPU 0: 14.75 GiB total capacity; 13.48 GiB already allocated; 14.81 MiB free; 13.65 GiB reserved in total by PyTorch) If reserved memory is >> allocated memory try setting max\_split\_size\_mb to avoid fragmentation. See documentation for Memory Management and PYTORCH\_CUDA\_ALLOC\_CONF

SEARCH STACK OVERFLOW

### Test the network

Test the trained network using the testset.

0초 오후 3:39에 완료됨

## ResNet152 의 OutOfMemory 에러

### ④ DenseNet

DenseNet 이 epoch10 만으로도 87%가 나왔다고 하시는 팀원분의 말을 듣고 마지막으로 DenseNet 을 조금 더 다뤄보기로 하였다. 169, 121 을 처음에 사용해 보았는데 둘 다 OutOfMemory 에러가 나서 121 로 캐시 비우는 코드를 추가한 후에 에폭 10 또는 15, 배치사이즈를 64 로 해보았는데 80 후반대의 좋은 성능이 나왔다. 하지만 용량의 한계로 에폭을 늘리지 못해서 그런지 80 후반에서 더 이상 오르지 않았다. 또한 복잡한 레이어 구조를 가지고 있어서 그런지 러닝타임이 1 시간 정도로 매우 길었다.

```
[ ] # As usual, a bit of setup
    # If you need other libraries, you should import the libraries.

import os, sys
import torch
from torch import nn
from torch.nn import functional as F

import torchvision
import torchvision.transforms as transforms
import torchvision.datasets as datasets

import matplotlib.pyplot as plt
import numpy as np

import math
import gc

# Set the device
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)

gc.collect()
torch.cuda.empty_cache()
```

+ 코드 + 텍스트

<> ▾ Data Loader

## GPU 캐시비우는 코드 추가

```
class Bottleneck(nn.Module):
    def __init__(self, in_planes, growth_rate):
        super(Bottleneck, self).__init__()
        self.bn1 = nn.BatchNorm2d(in_planes)
        self.conv1 = nn.Conv2d(in_planes, 4*growth_rate, kernel_size=1, bias=False)
        self.bn2 = nn.BatchNorm2d(4*growth_rate)
        self.conv2 = nn.Conv2d(4*growth_rate, growth_rate, kernel_size=3, padding=1, bias=False)

    def forward(self, x):
        out = self.conv1(F.relu(self.bn1(x)))
        out = self.conv2(F.relu(self.bn2(out)))
        out = torch.cat([out, x], 1)
        return out

class Transition(nn.Module):
    def __init__(self, in_planes, out_planes):
        super(Transition, self).__init__()
        self.bn = nn.BatchNorm2d(in_planes)
        self.conv = nn.Conv2d(in_planes, out_planes, kernel_size=1, bias=False)

    def forward(self, x):
        out = self.conv(F.relu(self.bn(x)))
        out = F.avg_pool2d(out, 2)
        return out

class DenseNet(nn.Module):
    def __init__(self, block, nblocks, growth_rate=12, reduction=0.5, num_classes=10):
        super(DenseNet, self).__init__()

class DenseNet(nn.Module):
    def __init__(self, block, nblocks, growth_rate=12, reduction=0.5, num_classes=10):
        super(DenseNet, self).__init__()
        self.growth_rate = growth_rate

        num_planes = 2*growth_rate
        self.conv1 = nn.Conv2d(3, num_planes, kernel_size=3, padding=1, bias=False)

        self.dense1 = self._make_dense_layers(block, num_planes, nblocks[0])
        num_planes += nblocks[0]*growth_rate
        out_planes = int(math.floor(num_planes*reduction))
        self.trans1 = Transition(num_planes, out_planes)
        num_planes = out_planes

        self.dense2 = self._make_dense_layers(block, num_planes, nblocks[1])
        num_planes += nblocks[1]*growth_rate
        out_planes = int(math.floor(num_planes*reduction))
        self.trans2 = Transition(num_planes, out_planes)
        num_planes = out_planes

        self.dense3 = self._make_dense_layers(block, num_planes, nblocks[2])
        num_planes += nblocks[2]*growth_rate
        out_planes = int(math.floor(num_planes*reduction))
        self.trans3 = Transition(num_planes, out_planes)
        num_planes = out_planes

        self.dense4 = self._make_dense_layers(block, num_planes, nblocks[3])
        num_planes += nblocks[3]*growth_rate

        self.bn = nn.BatchNorm2d(num_planes)
        self.linear = nn.Linear(num_planes, num_classes)

    def _make_dense_layers(self, block, in_planes, nblocks):
```

```
layers.append(block(in_planes, self.growth_rate))
in_planes += self.growth_rate
return nn.Sequential(*layers)

def forward(self, x):
    out = self.conv1(x)
    out = self.trans1(self.dense1(out))
    out = self.trans2(self.dense2(out))
    out = self.trans3(self.dense3(out))
    out = self.dense4(out)
    out = F.avg_pool2d(F.relu(self.bn(out)), 4)
    out = out.view(out.size(0), -1)
    out = self.linear(out)
    return out

def DenseNet121():
    return DenseNet(Bottleneck, [6,12,24,16], growth_rate=32)

def DenseNet169():
    return DenseNet(Bottleneck, [6,12,32,32], growth_rate=32)

def DenseNet201():
    return DenseNet(Bottleneck, [6,12,48,32], growth_rate=32)

def DenseNet161():
    return DenseNet(Bottleneck, [6,12,36,24], growth_rate=48)

net = DenseNet121().to(device)

# Loss function and optimizer
loss_fun = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.001)
#stepLR
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)

- Train the network

Train your own network using the above loss function and optimizer.

[ ] # Train the model
epochs = 10 # number of epochs

for epoch in range(epochs):
    loss_tmep = 0.0
    epoch_loss = 0.0

    for i, data in enumerate(trainloader, start=0):
        # Load the data
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)

        # Estimate the output using the network
        outputs = net(inputs)

# Test the trained model with overall test dataset

correct = 0
total = 0
for data in testloader:
    # Load the data
    inputs_test, labels_test = data
    inputs_test = inputs_test.to(device)
    labels_test = labels_test.to(device)

    # Estimate the output using the trained network
    outputs_test = net(inputs_test)
    _, predicted = torch.max(outputs_test.data, 1)

    # Calculate the accuracy
    total += labels_test.size(0)
    correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

## [SimpleNet / Training 5 epochs] Accuracy of the network on the 10,000 test images: 9 %
## [VGG11 / Training 5 epochs] Accuracy of the network on the 10,000 test images: 11 %
[+] Accuracy of the network on the 10,000 test images: 88 %
```

DenseNet121 88%

아무래도 ResNet 과 DenseNet 은 GPU 의 한계와 시간이 너무 많이 걸리는 관계로 에폭 사이즈를 많이 늘리지 못해서 작은 에폭으로도 80 후반의 성능이 나오긴 하지만 그 이상의 성과를 기대하기는 어려웠다. 이런 면에서 vgg 가 가장

적합했던 것 같다. 시간도 적게 걸리고 에폭을 늘려도 gpu 소모가 크지 않고 다양한 조합을 시도해 볼 수 있기 때문이다.

## II. Data Processing (Normalization, Augmentation, Initialization)

### ① Normalization

초반에 VGG 모델에서 `normalize(0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)`로 바꾸어 봤는데 성능이 떨어지거나 별 차이가 없었다.

```
transforms_cifar10 = transforms.Compose([transforms.Resize((32, 32)),
                                         transforms.ToTensor(),
                                         transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))])

# Train dataset
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transforms_cifar10)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, num_workers=2)

# Test dataset
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transforms_cifar10)
testloader = torch.utils.data.DataLoader(testset, batch_size=128, shuffle=False, num_workers=2)

# Classes of CIFAR-10 dataset
classes = ("plane", "car", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck")
```

그래서 인터넷과 ChatGPT 를 통해 검색해본 결과 CIFAR-10 데이터셋에 가장 적합한 정규화(normalization) 값은 일반적으로 평균(mean)을 0 으로, 표준 편차(standard deviation)를 1 로 조정하는 것이며 CIFAR-10 은 픽셀의 값이 0 부터 255 까지의 범위를 가지는 RGB 이미지로 구성되어 있기 때문에 평균을 0 으로 조정하기 위해서 255 로 나누고, 표준 편차를 1 로 조정하기 위해서 255 로 나눈 값을 사용하는 것이 좋다고 한다. 따라서 CIFAR-10 데이터셋을 정규화 할 때는 평균(Mean): [0.5, 0.5, 0.5], 표준 편차(Standard Deviation): [0.5, 0.5, 0.5]로 설정하면 RGB 의 평균이 0.5 로, 표준 편차가 0.5 로 조정되어 더욱 좋은 결과가 나올 수 있다고 한다. 그래서 Normalization 값은 바꾸지 않고 다시 0.5 로 설정하여 진행하였다.

### ② Augmentation

Augmentation 을 초기에 아래 사진과 같이 추가했는데, Augmentation 적용이 안되고 있었다. `transforms_cifar10` 부분을 `transform_train` 으로 바꾸어야 적용이 된다는 것을 나중에 발견하였다. 따라서 DenseNet 을 실습하면서 해당 부분을 수정하였는데 성능이 오히려 떨어졌었다.

```
transforms.ToTensor(),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.5, contrast=0.3, saturation=0.2, hue=0.3),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Train dataset
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=256, shuffle=True, num_workers=2)

# Test dataset
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_train)
testloader = torch.utils.data.DataLoader(testset, batch_size=256, shuffle=False, num_workers=2)

# Classes of CIFAR-10 dataset
classes = ("plane", "car", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck")

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:01<00:00, 45066709.781it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified

[4] # Examples of dataset
```

## Augmentation 초기 적용 모습

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.5, contrast=0.3, saturation=0.2, hue=0.3),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Train dataset
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True, num_workers=2)

# Test dataset
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_train)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False, num_workers=2)

# Classes of CIFAR-10 dataset
classes = ("plane", "car", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck")

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:13<00:00, 12630745.51it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified

[ ] # Examples of dataset
```

## Augmentation 적용을 다시 수정한 모습

```
transforms_cifar10 = transforms.Compose([transforms.Resize((32, 32)), #channel 32x32
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.5, contrast=0.3, saturation=0.2, hue=0.3),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Train dataset
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True, num_workers=2)

# Test dataset
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_train)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False, num_workers=2)

# Classes of CIFAR-10 dataset
classes = ("plane", "car", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck")

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:13<00:00, 12630745.51it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified

[ ] # Examples of dataset
```

```

self.bn = nn.BatchNorm2d(in_planes)
self.conv = nn.Conv2d(in_planes, out_planes, kernel_size=1, bias=False)

def forward(self, x):
    out = self.conv(F.relu(self.bn(x)))
    out = F.avg_pool2d(out, 2)
    return out

def initialize_weights(m):
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_uniform_(m.weight.data, nonlinearity='relu')
        if m.bias is not None:
            nn.init.constant_(m.bias.data, 0)
    elif isinstance(m, nn.BatchNorm2d):
        nn.init.constant_(m.weight.data, 1)
        nn.init.constant_(m.bias.data, 0)
    elif isinstance(m, nn.Linear):
        nn.init.kaiming_uniform_(m.weight.data)
        nn.init.constant_(m.bias.data, 0)

class DenseNet(nn.Module):
    def __init__(self, block, nblocks, growth_rate=12, reduction=0.5, num_classes=10):
        super(DenseNet, self).__init__()
        self.growth_rate = growth_rate

        num_planes = 2 * growth_rate
        self.conv1 = nn.Conv2d(3, num_planes, kernel_size=3, padding=1, bias=False)

        self.dense1 = self._make_dense_layers(block, num_planes, nblocks[0])
        num_planes += nblocks[0] * growth_rate
        out_planes = int(math.floor(num_planes * reduction))

```

## Loss function and optimizer

Set the **loss function and optimizer** for training CNN. You can modify the loss function or optimizer for better performance.

```

[ ] # Loss function and optimizer
loss_fun = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.001)

#StepLR
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)

```

## Train the network

Train your own network using the above loss function and optimizer.

```

[ ] # Train the model
epochs = 15 # number of epochs

for epoch in range(epochs):

    loss_tmp = 0.0
    epoch_loss = 0.0

    for i, data in enumerate(trainloader, start=0):
        # Load the data
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)

```

## # Test the trained model with overall test dataset

```

correct = 0
total = 0
for data in testloader:
    # Load the data
    inputs_test, labels_test = data
    inputs_test = inputs_test.to(device)
    labels_test = labels_test.to(device)

    # Estimate the output using the trained network
    outputs_test = net(inputs_test)
    _, predicted = torch.max(outputs_test.data, 1)

    # Calculate the accuracy
    total += labels_test.size(0)
    correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

## [SimpleNet / Training 5 epochs] Accuracy of the network on the 10,000 test images: 9 %
## [VGG11 / Training 5 epochs] Accuracy of the network on the 10,000 test images: 11 %

```

➤ Accuracy of the network on the 10,000 test images: 76 %

DenseNet121 에서 Augmentation 과 Initialization 을 넣었을 때 정확도가 76%로 감소한 모습

아마도 DenseNet 에서는 에폭 수가 작아서 Augmentation 을 넣으면 과적합(Overfitting)과 원본 데이터 왜곡이 더 도드라지게 일어나서 성능이 떨어지는 것 같다.

그래서 마지막으로 VGG19 모델 에폭 100 으로 vertical flip 부분을 지우고 brightness, contrast, saturation, hue 값을 0.1 로 줄여서 적용했더니 아래 사진과 같이 성능이 92%로 오르는 것을 확인할 수 있었다.

```
transforms_cifar10 = transforms.Compose([transforms.Resize((32, 32)),
                                         transforms.ToTensor(),
                                         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                                         ])

# Data augmentation
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Train dataset (with data augmentation)
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=256, shuffle=True, num_workers=2)

# Test dataset (without data augmentation)
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transforms_cifar10)
testloader = torch.utils.data.DataLoader(testset, batch_size=256, shuffle=False, num_workers=2)

# Classes of CIFAR-10 dataset
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:03<00:00, 51249726.14it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified

[9] # Test the trained model with overall test dataset

correct = 0
total = 0

with torch.no_grad(): # Disable gradient tracking
    net.eval()
    for data in testloader:
        # Load the data
        inputs_test, labels_test = data
        inputs_test = inputs_test.to(device)
        labels_test = labels_test.to(device)

        # Estimate the output using the trained network
        outputs_test = net(inputs_test)
        _, predicted = torch.max(outputs_test.data, 1)

        # Calculate the accuracy
        total += labels_test.size(0)
        correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

Accuracy of the network on the 10,000 test images: 92 %
```

### ③ Initialization

“He 초기화 (He Initialization): He 초기화는 ReLU(렐루)와 같은 활성화 함수를 사용하는 신경망에 적합한 초기화 방법입니다. He 초기화는 가중치를 랜덤하게 초기화할 때, 표준 편차를  $\sqrt{2/n}$ 으로 설정하여 초기화합니다.”

chatGPT 에서 다음과 같은 내용을 찾아보고 우리는 ReLU 를 사용하기 때문에 He Initialization 을 추가하기로 하였다. 하지만 처음에는 적용이 잘 안되었는지 효과가 좋지 않았다. 특히 DenseNet 에서는 에폭 수가 적어서 그런지 성능이 확 떨어졌다.

```
self.bn = nn.BatchNorm2d(in_planes)
self.conv = nn.Conv2d(in_planes, out_planes, kernel_size=1, bias=False)

def forward(self, x):
    out = self.conv(F.relu(self.bn(x)))
    out = F.avg_pool2d(out, 2)
    return out

def initialize_weights(m):
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_uniform_(m.weight.data, nonlinearity='relu')
    if m.bias is not None:
        nn.init.constant_(m.bias.data, 0)
    elif isinstance(m, nn.BatchNorm2d):
        nn.init.constant_(m.weight.data, 1)
        nn.init.constant_(m.bias.data, 0)
    elif isinstance(m, nn.Linear):
        nn.init.kaiming_uniform_(m.weight.data)
        nn.init.constant_(m.bias.data, 0)

class DenseNet(nn.Module):
    def __init__(self, block, nblocks, growth_rate=12, reduction=0.5, num_classes=10):
        super(DenseNet, self).__init__()
        self.growth_rate = growth_rate

        num_planes = 2 * growth_rate
        self.conv1 = nn.Conv2d(3, num_planes, kernel_size=3, padding=1, bias=False)

        self.dense1 = self._make_dense_layers(block, num_planes, nblocks[0])
        num_planes += nblocks[0] * growth_rate
        out_planes = int(math.floor(num_planes * reduction))

# Test the trained model with overall test dataset

correct = 0
total = 0
for data in test_loader:
    # Load the data
    inputs_test, labels_test = data
    inputs_test = inputs_test.to(device)
    labels_test = labels_test.to(device)

    # Estimate the output using the trained network
    outputs_test = net(inputs_test)
    _, predicted = torch.max(outputs_test.data, 1)

    # Calculate the accuracy
    total += labels_test.size(0)
    correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

## [SimpleNet / Training 5 epochs] Accuracy of the network on the 10,000 test images: 9 %
## [VGG11 / Training 5 epochs] Accuracy of the network on the 10,000 test images: 11 %

➡ Accuracy of the network on the 10,000 test images: 76 %
```

DenseNet 에서 처음 He Initialization 을 추가하였더니 정확도가 줄어든 모습



이후에 이 부분도 Augmentation 과 마찬가지로 net.eval() 함수를 적용하였더니 해결되었다. 확실히 Augmentation 과 Initialization 은 넣는 것이 정확도 향상에 도움이 되며 eval 함수를 사용하여 평가모드로 진행하는 것이 효과가 좋은 것 같다.

### III. Optimizer/Scheduler

옵티마이저는 SGD 와 비교해본 결과 Adam 이 더 좋았던 것 같다. Adam 은 momentum 이 없어서 betas 나 weight decay 를 사용하며, Adam 옵티마이저에 적절한 betas 값은 일반적으로 기본값인 0.9 와 0.999 를 사용하는 것이 좋다고 한다.

```

# Loss function and optimizer
Set the loss function and optimizer for training CNN. You can modify the loss function or optimizer for better performance.

# Loss function and optimizer
loss_fun = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
optimizer = torch.optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)

# Train the network
Train your own network using the above loss function and optimizer.

[15] # Train the model
epochs = 30 # number of epochs

for epoch in range(epochs):

    loss_tmp = 0.0
    epoch_loss = 0.0

    for i, data in enumerate(trainloader, start=0):
        # Load the data
        inputs, labels = data
        inputs = inputs.to(device)

# Test the trained model with overall test dataset

correct = 0
total = 0
for data in testloader:
    # Load the data
    inputs_test, labels_test = data
    inputs_test = inputs_test.to(device)
    labels_test = labels_test.to(device)

    # Estimate the output using the trained network
    outputs_test = net(inputs_test)
    _, predicted = torch.max(outputs_test.data, 1)

    # Calculate the accuracy
    total += labels_test.size(0)
    correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % ((100 * correct / total)))

## [SimpleNet / Training 5 epochs] Accuracy of the network on the 10,000 test images: 9 %
## [VGG11 / Training 5 epochs] Accuracy of the network on the 10,000 test images: 11 %

Accuracy of the network on the 10,000 test images: 80 %
```

SGD 80%

```
▼ Loss function and optimizer
Set the loss function and optimizer for training CNN. You can modify the loss function or optimizer for better performance.

[6] # Loss function and optimizer
    loss_fun = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=0.001, betas=(0.9, 0.999))
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=3, verbose=True)

▼ Train the network
Train your own network using the above loss function and optimizer.

# Train the model
epochs = 100 # number of epochs

[9] # Test the trained model with overall test dataset
    correct = 0
    total = 0

    with torch.no_grad(): # Disable gradient tracking
        net.eval()
        for data in test_loader:
            # Load the data
            inputs_test, labels_test = data
            inputs_test = inputs_test.to(device)
            labels_test = labels_test.to(device)

            # Estimate the output using the trained network
            outputs_test = net(inputs_test)
            _, predicted = torch.max(outputs_test.data, 1)

            # Calculate the accuracy
            total += labels_test.size(0)
            correct += (predicted == labels_test).sum()

    # Final accuracy
    print('Accuracy of the network on the 10,000 test images: %d %%' % ((100 * correct / total)))

Accuracy of the network on the 10,000 test images: 92 %
```

Adam 92%

스케줄러는 팀원들끼리 10 개의 스케줄러를 검색하여 이를 나눠서 사용해보았는데 스케줄러의 차이만 확인하기 위해 모델은 vgg19, 에폭 30 과 배치사이즈 256 으로 고정해서 진행하였다. 나는 그중에서 LambdaLR, MultiplicativeLR, StepLR, MultiStepLR 4 개를 맡았다. 그랬더니 결과가 다음과 같았다.

LambdaLR 82%

MultiplicativeLR 85%

StepLR 87%

MultiStepLR 86%

```
Set the loss function and optimizer for training CNN. You can modify the loss function or optimizer for better performance.

[0] # Loss function and optimizer
    loss_fun = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=0.0001)
    #scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

- Train the network

Train your own network using the above loss function and optimizer.

# Train the model
epochs = 20 # number of epochs

for epoch in range(epochs):

    loss_tmp = 0.0
    epoch_loss = 0.0

    for i, data in enumerate(trainloader, start=0):
        # Load the data
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)

        # Estimate the output using the network
        outputs = net(inputs)

# Test the trained model with overall test dataset

correct = 0
total = 0
for data in testloader:
    # Load the data
    inputs_test, labels_test = data
    inputs_test = inputs_test.to(device)
    labels_test = labels_test.to(device)

    # Estimate the output using the trained network
    outputs_test = net(inputs_test)
    _, predicted = torch.max(outputs_test.data, 1)

    # Calculate the accuracy
    total += labels_test.size(0)
    correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

## [SimpleNet / Training 5 epochs] Accuracy of the network on the 10,000 test images: 9 %
## [VGG11 / Training 5 epochs] Accuracy of the network on the 10,000 test images: 11 %

Accuracy of the network on the 10,000 test images: 75 %
```

스케줄러 없는 경우 75%

다른 팀원의 결과는 다음과 같았다.

ExponentialLR 70%

CosineAnnealingLR 78%

ReduceLROnPlateau 76%

또 다른 팀원분은 CyclicLR, CyclicLR, CosineAnnealingWarmRestarts 3 개를 해봤는데 눈에 띄는 결과가 없어서 그냥 무시하기로 하였다. 그리고 다양한 자료들을 찾아본 결과 CIFAR10 에는 StepLR, LambdaLR, ReduceLROnPlateau 가 자주 사용되어서 스케줄러는 세 가지 정도로 추리기로 하였다.

stepLR 은 아래 사진과 같이 평균적으로 좋은 성능을 내긴 하지만 80 후반에서 더 이상 오르지 않았다.

```
cfg = {'VGG19': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 512, 512, 'M']}

class VGG19(nn.Module):
    def __init__(self):
        super(VGG19, self).__init__()
        self.features = self.make_layers(cfg['VGG19'])
        self.classifier = nn.Linear(512, 10)

    def make_layers(self, cfg):
        layers = []
        in_channels = 3
        for x in cfg:
            if x == 'M':
                layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
            else:
                layers += [nn.Conv2d(in_channels, x, kernel_size=3, padding=1),
                            nn.BatchNorm2d(x),
                            nn.ReLU(inplace=True)]
                in_channels = x
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.features(x)
        out = out.view(out.size(0), -1)
        out = self.classifier(out)
        return out

net = VGG19().to(device)

# Loss function and optimizer
loss_fun = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.001)

#stepLR
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

#lambdaLR
#scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer=optimizer, lr_lambda=lambda epoch: 0.95 ** epoch)

#MultiplicativeLR
#scheduler = torch.optim.lr_scheduler.MultiplicativeLR(optimizer=optimizer, lr_lambda=lambda epoch: 0.95 ** epoch)

#MultiStepLR
#scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[30, 80], gamma=0.5)

# Train the network
Train your own network using the above loss function and optimizer.

[18] # Train the model
epochs = 30 # number of epochs

for epoch in range(epochs):
    loss_tmp = 0.0

# Test the trained model with overall test dataset

correct = 0
total = 0
for data in testloader:
    # Load the data
    inputs_test, labels_test = data
    inputs_test = inputs_test.to(device)
    labels_test = labels_test.to(device)

    # Estimate the output using the trained network
    outputs_test = net(inputs_test)
    _, predicted = torch.max(outputs_test.data, 1)

    # Calculate the accuracy
    total += labels_test.size(0)
    correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

## [SimpleNet / Training 5 epochs] Accuracy of the network on the 10,000 test images: 9 %
## [VGG11 / Training 5 epochs] Accuracy of the network on the 10,000 test images: 11 %

Accuracy of the network on the 10,000 test images: 87 %
```

그래서 아래 사진과 같이 ReduceLROnPlateau 에서 인자를 몇 개 줄이고 적용해보았더니 예폭이 100 으로 커져서 그런지 효과가 좋았던 것 같다. ReduceLROnPlateau 는 주어진 횟수의 epoch 동안 loss function 이 개선되지 않을 때 학습률을 조절해준다. 예폭이 클수록 loss 가 줄지 않고 정체되는 경우가 많은 것 같아서 이때 이 스케줄러가 큰 역할을 하는 것 같았다.

```
[ ] # Loss function and optimizer
loss_fun = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=0.001, betas=(0.9, 0.999))
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=3, verbose=True)

- Train the network

Train your own network using the above loss function and optimizer.

# Train the model
epochs = 100 # number of epochs

for epoch in range(epochs):

    loss_tmp = 0.0
    epoch_loss = 0.0

    for i, data in enumerate(trainloader, start=0):
        # Load the data
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)

        # Estimate the output using the network
        outputs = net(inputs)

        # Calculate the loss between the output of the network and label

[ ] [Epoch - 86] Loss: 0.004
[Epoch - 87] Loss: 0.005
[Epoch - 88] Loss: 0.004
[Epoch - 89] Loss: 0.004
[Epoch - 90] Loss: 0.003
[Epoch - 91] Loss: 0.004
[Epoch - 92] Loss: 0.004
[Epoch - 93] Loss: 0.004
[Epoch - 94] Loss: 0.003
[Epoch - 95] Loss: 0.003
[Epoch - 96] Loss: 0.004
[Epoch - 97] Loss: 0.003
[Epoch - 98] Loss: 0.003
Epoch 00099: reducing learning rate of group 0 to 1.0000e-05.
[Epoch - 99] Loss: 0.004
[Epoch - 100] Loss: 0.003
Finished Training

- Test the network

Test the trained network using the testset.

Accuracy of the network on the 10,000 test images is the final accuracy of your network.

The closer the accuracy is to 100%, the better the network classifies the input image.

# Test the trained model with sample

dataloader_test = iter(testloader)
img_test, labels_test = next(dataloader_test)

imshow(torchvision.utils.make_grid(img_test))

correct = 0
total = 0

with torch.no_grad(): # Disable gradient tracking
    for data in testloader:
        # Load the data
        inputs_test, labels_test = data
        inputs_test = inputs_test.to(device)
        labels_test = labels_test.to(device)

        # Estimate the output using the trained network
        outputs_test = net(inputs_test)
        _, predicted = torch.max(outputs_test.data, 1)

        # Calculate the accuracy
        total += labels_test.size(0)
        correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

Accuracy of the network on the 10,000 test images: 91 %
```

## IV. Final Code Description

최종 제출 코드를 간단히 설명해보았다.

```
transforms_cifar10 = transforms.Compose([transforms.Resize((32, 32)),
                                         transforms.ToTensor(),
                                         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                                         ])

# Data augmentation
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Train dataset (with data augmentation)
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=256, shuffle=True, num_workers=2)

# Test dataset (without data augmentation)
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transforms_cifar10)
testloader = torch.utils.data.DataLoader(testset, batch_size=256, shuffle=False, num_workers=2)

# Classes of CIFAR-10 dataset
classes = ("plane", "car", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck")

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170488071/170488071 [00:03<00:00, 51249726.14it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

Augmentation 적용 시 ColerJitter 값 0.1 로 적용하여 과적합이 일어나지 않도록 적당한 값으로 줄여주었다.

```
# VGG19
cfg = {'VGG19': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 'M', 512, 512, 512, 512, 'M']}

class VGG19(nn.Module):
    def __init__(self):
        super(VGG19, self).__init__()
        self.features = self.make_layers(cfg['VGG19'])
        self.classifier = nn.Linear(512, 10)
        self.initialize_weights()

    def make_layers(self, cfg):
        layers = []
        in_channels = 3
        for x in cfg:
            if x == 'M':
                layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
            else:
                layers += [nn.Conv2d(in_channels, x, kernel_size=3, padding=1),
                           nn.BatchNorm2d(x),
                           nn.ReLU()]
                in_channels = x
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.features(x)
        out = out.view(out.size(0), -1)
        out = self.classifier(out)
        return out

    def initialize_weights(self):
```

VGG19 가 적은 GPU 로도 잘 돌아가고 에폭을 늘려도 시간이 오래 걸리지 않고 다양한 조합의 차이를 확인할 수 있어서 효율적이기 때문에 이 모델로 선정하였다.



```
[9] # Test the trained model with overall test dataset

correct = 0
total = 0

with torch.no_grad(): # Disable gradient tracking
    net.eval()
    for data in testloader:
        # Load the data
        inputs_test, labels_test = data
        inputs_test = inputs_test.to(device)
        labels_test = labels_test.to(device)

        # Estimate the output using the trained network
        outputs_test = net(inputs_test)
        _, predicted = torch.max(outputs_test.data, 1)

        # Calculate the accuracy
        total += labels_test.size(0)
        correct += (predicted == labels_test).sum()

# Final accuracy
print('Accuracy of the network on the 10,000 test images: %d %%' % (100 * correct / total))

Accuracy of the network on the 10,000 test images: 92 %
```

✓ 3초 오후 10:42에 완료됨

최종 결과로 정확도가 92%가 나왔다.

참고문헌: chatGPT, [\[PyTorch\] Weight Initialization \(기울기 초기화\) \(tistory.com\)](#), [\[PyTorch\] PyTorch 가 제공하는 Learning rate scheduler 정리 \(tistory.com\)](#), [GitHub 모델 별 코드 정리 https://github.com/kuangliu/pytorch-cifar/tree/master/models](#), [Weights & Biases \(wandb.ai\)](#),