

1

[SUM]실전인공지능으로 이어지는 파이토치 딥러닝

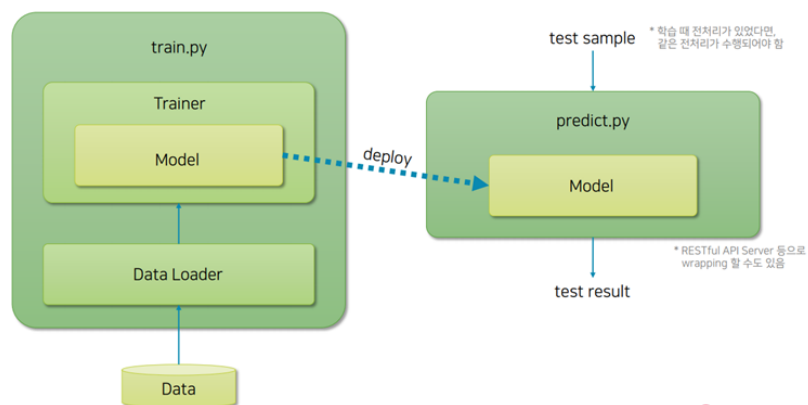
■ 강의 | [Inflearn] 실전인공지능으로 이어지는 파이토치 딥러닝

Contents

- 1) Section 02 | PyTorch 입문
- 2) Section 03 | Supervised Learning
- 3) Section 04 | Pretrained Model
- 4) Section 05 | GAN & K-Means Clustering
- 5) Section 06 | Visualization
- 6) Section 07 | Improvement
- 7) Section 08 | Practice

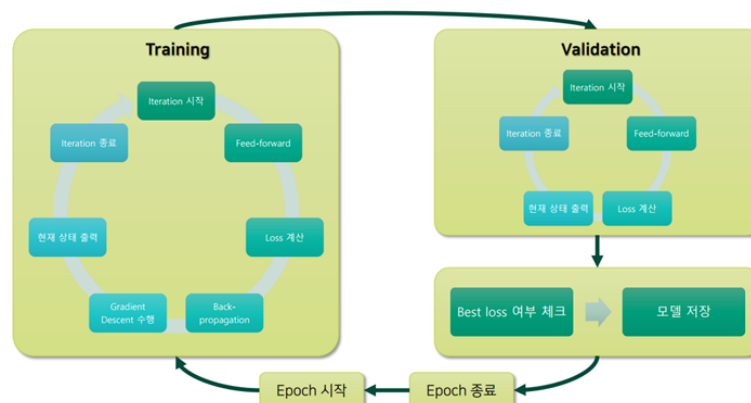
Section 02 | PyTorch 입문

[Deep Learning Overview]



- 실무적 관점에서의 DL Process
- train.py / main.py | data loader 및 trainer 를 통한 model 학습
- predict.py / test.py | model 을 통한 예측값 추출

[Deep Learning Algorithm]



- 매 Iteration 마다 feed-forward 를 통해 데이터가 이동하고 그로 인한 Loss 계산
- Loss 를 바탕으로 한 Back-Propagation 으로 얻어진 Gradient Descent 를 통해 Weight 조정

[Tensor 텐서]

- 파이토치의 기본 단위 \rightarrow list 혹은 array 대신 tensor 형태로 변환해서 사용
- GPU 연산 가능
- Numpy 배열과 유사하여 다루기 용이

\Rightarrow 3차원 이상의 행렬을 GPU 연산이 가능한 형태로 사용

- 텐서 만들기
 - rand 의 경우 가우시안 분포를 따르는 행렬 생성 ($N(0,1)$)

```
# 빈 텐서 생성

x = torch.empty(5,4) # 5x4 행렬 생성
print(x) # 초기화되지 않은 행렬인 경우 해당 시점에 할당된 메모리에 존재하던 값들이 초기값으로 나타난다.

torch.ones(3,3) # 3x3 일 행렬

torch.zeros(2) # 2행 영 벡터

torch.rand(5,6) # 5x6 랜덤 행렬
```

- 넘파이 배열 텐서로 변환
 - tensor() 를 활용해 넘파이 배열을 텐서로 변경

```
r = np.array([4,56,7]) # 넘파이 배열 생성

torch.tensor(r) #넘파이 배열을 텐서로 쉽게 변환할 수 있다.
```

- 텐서 연산

```
y.add(x) # y + x
y.add_(x) # y += x
```

- 텐서의 크기 변환

```
x = torch.rand(8,8)
a = x.view(64) # 크기를 바꿔주는 view 8x8 -> 64
print(a.size())

b = x.view(-1,4,4) # -1은 원래 크기(64)가 되게 하는 값 8x8 -> -1x4x4 즉, 4x4x4이다.
# -1의 경우 자동 지정

print(b.size())

# 따라서 -1은 원래 크기가 되게 하는 값이 자동으로 지정되기 때문에 한 번만 사용할 수 있다.
# 예를 들어 x.view(-1, -1,4)와 같은 선언은 오류가 난다.
```

- 단일 텐서에서 값으로 뽑아내기

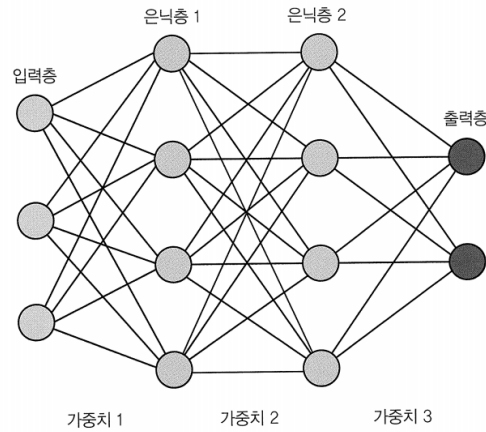
```
x = torch.ones(1) # 1으로 구성된 1행 짜리 텐서
print(x.item()) # 단일 텐서에서 value를 뽑아내는 방법

# 손실함숫값과 같이 숫자가 하나인 텐서를 텐서가 아닌 값으로 만들어 준다.
# 손실함숫값을 저장 시 텐서 형태 보다 item()으로 저장
```

[역전파]

: 인공 신경망을 최적화 하는 과정에서 미분은 필수인 요소인데 파이토치는 최적화 과정인 역전파를 쉽게 할 수 있도록 자동 미분 계산을 제공한다.

- 전파 (forward propagation)
 - : 입력값이 들어와 여러 개의 은닉층을 순서대로 거쳐 결괏값을 내는 과정
 - ▼ 예시) 2개의 은닉층을 가지는 인공 신경망



$$\begin{matrix}
 \begin{bmatrix} w_{00} & w_{01} & w_{02} & w_{03} \\ w_{10} & w_{11} & w_{12} & w_{13} \\ w_{20} & w_{21} & w_{22} & w_{23} \end{bmatrix} &
 \begin{bmatrix} w_{00} & w_{01} & w_{02} & w_{03} \\ w_{10} & w_{11} & w_{12} & w_{13} \\ w_{20} & w_{21} & w_{22} & w_{23} \\ w_{30} & w_{31} & w_{32} & w_{33} \end{bmatrix} &
 \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \\ w_{20} & w_{21} \\ w_{30} & w_{31} \end{bmatrix} \\
 \text{가중치1}(3 \times 4) & \text{가중치2}(4 \times 4) & \text{가중치3}(4 \times 2)
 \end{matrix}$$


→ 가중치 w , 편차 b , 활성화 함수 σ

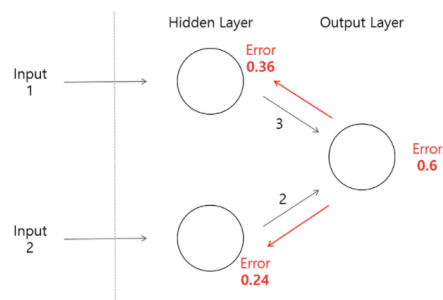
$$y^* = w_3 \times \sigma(w_2 \times \sigma(w_1 \times x + b_1) + b_2) + b_3$$

이렇게 예측값 y^* 를 구하는 과정이 바로 전파!

• 역전파 (backward propagation)

: 결과와 정답의 차이로 계산된 손실을 연쇄법칙(chain rule)을 이용하여 입력 단까지 다시 전달하는 과정

 Input이 들어오는 방향(순전파)으로 output layer에서 결과 값이 나온다. 결과값은 오차(error)를 가지게 되는데 역전파는 이 오차(error)를 다시 역방향으로 보내면서 가중치를 업데이트하여 output에서 발생했던 오차를 반영하는 것!



이렇게 가중치를 업데이트할 때 경사하강법을 이용!

$$w_{t+1} = w_t - \text{gradient} \times \text{learning rate}$$

$$\text{loss} = y^* - y = w_3 \times \text{sig}(w_2 \times \text{sig}(w_1 \times x + b_1) + b_2) + b_3 - y$$

$$\frac{\partial \text{loss}}{\partial w_3} = \text{sig}(w_2 \times \text{sig}(w_1 \times x + b_1) + b_2)$$

$$\frac{\partial \text{loss}}{\partial b_3} = 1$$

$$\frac{\partial \text{loss}}{\partial w_2} = ?$$

◦ Chain Rule

$$h(x) = f(g(x))$$

$$h'(x) = f'(g(x))g'(x)$$



$$\frac{\partial \text{loss}}{\partial w_2} = w_3 \times \text{sigmoid}'(h_{2_in}) \times \text{sigmoid}(w_1 \times x + b_1)$$

```
h2_in = w2*sig(w1*x + b1) + b2
```

[Pytorch 코드]

⇒ `requires_grad=True` 로 선언된 변수를 기준으로 `gradient`가 전달

```
# requires_grad=True는 해당 텐서를 기준으로 모든 연산들을 추적할 수 있게 하는 옵션이다.
# 즉, x에 대해서 연쇄 법칙을 이용한 미분이 가능하다는 것이다.
# (순전파 시의 모든 연산이 추적)
x = torch.ones(2,2, requires_grad=True)
print(x)

# y는 x에 대한 식, z는 y에 대한 식, res는 z에 대한 식이다.
# 따라서 이는 합성함수의 개념으로써 x에 대해서 표현 및 미분이 가능하다.
y = x+1
z = 2*y**2
res = z.mean()
print("y: ", y)
print("z: ", z)
print("Result: ", res)
# grad_fn=...은 추적이 잘 되고 있다는 의미

# 역전파
res.backward() # res를 기준으로 역전파를 진행하겠다는 의미다.

# 역으로 식을 써내려 가보자.
# res = (z_1 + ... + z_4)/4
# z_i = 2 * y_i **2
# z_i = 2(x_i+1)**2
# d(res)/dx_i = x_i + 1

print(x)
print(x.grad)
# x.grad는 backward()가 선언 된 변수를 기준으로 미분을 한다. 즉 d(res)/dx를 계산한다.
# #d(res)/dx_i = x_i + 1
```

$$x = \begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \end{pmatrix}, y = x + 1, z = 2y^2 = 2(x + 1)^2$$

$$res = \frac{z_1 + z_2 + z_3 + z_4}{4}, \frac{\partial z}{\partial x} = 4(x + 1)$$

$$\begin{aligned} \frac{\partial res}{\partial x_i} &= \frac{\partial res}{\partial z_1} \frac{\partial z_1}{\partial x_i} + \frac{\partial res}{\partial z_2} \frac{\partial z_2}{\partial x_i} + \frac{\partial res}{\partial z_3} \frac{\partial z_3}{\partial x_i} + \frac{\partial res}{\partial z_4} \frac{\partial z_4}{\partial x_i} \\ &= \frac{1}{4} \left(\frac{\partial z_1}{\partial x_i} + \frac{\partial z_2}{\partial x_i} + \frac{\partial z_3}{\partial x_i} + \frac{\partial z_4}{\partial x_i} \right) \\ &= x_i + 1 \end{aligned}$$

[데이터 불러오기]

- 파이토치 제공 데이터를 사용하는 경우

```
# https://pytorch.org/docs/stable/torchvision/transforms.html
# 다양한 전처리 방법들을 확인할 수 있다.
# tr.Compose 내에 원하는 전처리를 차례대로 넣어주면 된다.

transf = tr.Compose([tr.Resize(16), tr.ToTensor()])
# 16x16으로 이미지 크기 변환 후 텐서 타입으로 변환한다.

# https://pytorch.org/docs/stable/torchvision/datasets.html
```

```
# 다양한 이미지 데이터셋을 확인할 수 있다.

# torchvision.datasets에서 제공하는 CIFAR10 데이터를 불러온다.
# root에는 다운로드 받을 경로를 입력한다.
# train=True이면 학습 데이터를 불러오고 train=False이면 테스트 데이터를 불러온다.
# 미리 선언한 전처리를 사용하기 위해 transform=transf를 작성한다.

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transf)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transf)
```

```
# DataLoader는 데이터를 미니 배치 형태로 만들어 준다.
# 따라서 배치 사이즈 및 셔플 여부 등을 선택할 수 있다.
trainloader = DataLoader(trainset, batch_size=50, shuffle=True)
testloader = DataLoader(testset, batch_size=50, shuffle=False)

# test data 에 batch 를 거는 이유
# 동패로 사용할 경우 메모리 과다 사용
```

- 정형화되지 않은 커스텀 데이터를 사용하는 경우

```
"""
from torch.utils.data import Dataset

class MyDataset(Dataset):

    def __init__(self):

    def __getitem__(self, index):

    def __len__(self):

이 양식을 통으로 가지고 다니자!!
"""

class TensorData(Dataset):

    def __init__(self, x_data, y_data):
        self.x_data = torch.FloatTensor(x_data) # 이미지 데이터를 FloatTensor로 변형
        self.x_data = self.x_data.permute(0,3,1,2) # (이미지 수)x(너비)x(높이)x(채널 수) -> (배치 크기)x(채널 수)x(너비)x(높이)
        self.y_data = torch.LongTensor(y_data) # 라벨 데이터를 LongTensor로 변형
        self.len = self.y_data.shape[0] # 클래스 내의 들어 온 데이터 개수

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index] # 뽑아 낸 데이터를 적어준다.

    def __len__(self):
        return self.len # 클래스 내의 들어 온 데이터 개수

# 파이토치에서는 (배치 크기)x(채널 수)x(너비)x(높이) 데이터가 사용
# 따라서 원래 데이터 (이미지 수)x(너비)x(높이)x(채널 수)를 변경해야만 한다.
# permute 는 0(이미지 수), 1(너비), 2 (높이), 3(채널 수)
# => 0(이미지 수), 3(채널 수), 1(너비), 2 (높이)로 변경하는 역할
# .permute(0,3,1,2)를 사용하는 것이다.
```

Section 03 | Supervised Learning (지도학습)

[인공신경망]

: 사람의 신경망을 모사하여 만든 예측 도구이다. 기본적으로 하나의 레이어에 다수의 노드를 가지고 있으며 여러 개의 레이어가 쌓인 신경망을 깊은 신경망이라고 한다. 이 때, 깊은 신경망을 이용하여 모델을 학습 시키는 방법을 딥러닝이라고 한다.

- 집 값 예측하기 | 다층신경망(MLP)

1. 기본 라이브러리 및 보스턴 데이터 불러오기

```
# import library

from sklearn.datasets import load_boston # 보스턴 집 값 데이터 불러오기
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split # 전체 데이터를 학습 데이터와 평가 데이터로 나누기
from sklearn.preprocessing import MinMaxScaler # 데이터 내의 값을 0이상 1이하의 값으로 scaling

# ANN
import torch
from torch import nn, optim # torch 내의 세부적인 기능을 불러오기 (신경망 기술, 손실함수, 최적화 방법 등)
from torch.utils.data import DataLoader, Dataset # 데이터를 모델에 사용할 수 있도록 정리해주는 라이브러리
import torch.nn.functional as F # torch 내의 세부적인 기능을 불러온다. (신경망 기술 등)

# Loss
from sklearn.metrics import mean_squared_error # Regression 문제의 평가를 위해 MSE를 불러옴

# Plot
import matplotlib.pyplot as plt
```

```
bos = load_boston()
```

```
# bos 내에는 data, feature name 등 다양한 정보를 포함하고 있음.
df = pd.DataFrame(bos.data) # bos 변수를 데이터프레임으로 생성
df.columns = bos.feature_names # bos 내에 저장되어 있는 feature name을 변수명에 저장
df['Price'] = bos.target # 집 값에 해당하는 타겟 값 호출

# 데이터프레임 확인
df.head(10)
```

⇒ 보스턴 데이터를 이용한 보스턴 집 값 예측 논문의 성능인 'RMSE = 0.08019'와 성능 비교

2. 데이터 스케일링(MinMax Scaling)

: 변수마다 각각의 scale 이 다르므로 열 기준으로 0과 1사이의 숫자로 바꿔줌.

```
# 데이터를 넘파이 배열로 만들기
X = df.drop('Price', axis=1).to_numpy() # 데이터프레임 -> 타겟값(Price)를 제외한 넘파이 배열로 만들기
Y = df['Price'].to_numpy().reshape((-1,1)) # 데이터프레임 -> 2차원의 넘파이 배열

# 데이터 스케일링
# sklearn에서 제공하는 MinMaxScaler
# 적용되는 식: (X-min(X))/(max(X)-min(X))

scaler = MinMaxScaler()

scaler.fit(X)
X = scaler.transform(X)

# 0과 1사이 범위로 맞춰주기
scaler.fit(Y)
Y = scaler.transform(Y)

# 타겟값까지 스케일링을 할 필요는 없지만 논문과 조건을 동일하게 하기 위해 스케일링 진행
```

3. 텐서 데이터와 배치 만들기

```
# 텐서 데이터로 변환하는 클래스(섹션2 참고)
class TensorData(Dataset):

    def __init__(self, x_data, y_data):
        self.x_data = torch.FloatTensor(x_data)
        self.y_data = torch.FloatTensor(y_data)
        self.len = self.y_data.shape[0]

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.len
```

```
# 전체 데이터를 학습 데이터와 평가 데이터로 분리
# 논문이 전체 데이터를 50:50으로 나뉘기에 test size를 0.5로 설정
## seed를 고정 안해줘서 항상 같은 결과가 나오지는 않을 것!

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.5)

# 학습 데이터, 시험 데이터 배치 형태로 구축
trainsets = TensorData(X_train, Y_train)
trainloader = DataLoader(trainsets, batch_size=32, shuffle=True)

testsets = TensorData(X_test, Y_test)
testloader = DataLoader(testsets, batch_size=32, shuffle=True)
```

4. 모델 구축

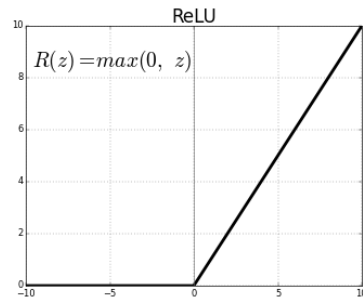
: 모델은 Regressor로 정의하며 입력층(노드 13개), 2개의 은닉층(노드 50개, 30개), 출력층(노드 1개)으로 구성된다. 데이터의 변수는 13개가 되고, 출력층은 집 값인 단일 값을 추출하는 것이므로 1개가 되는 것. (은닉층에 대해서는 실험을 하며 튜닝 가능)

```
class Regressor(nn.Module):
    def __init__(self):
        super().__init__() # 모델의 연산 정의하기
        self.fc1 = nn.Linear(13, 50, bias=True) # 입력층(13개) -> 은닉층1(50)으로 가는 연산
        self.fc2 = nn.Linear(50, 30, bias=True) # 은닉층1(50) -> 은닉층2(30)으로 가는 연산
        self.fc3 = nn.Linear(30, 1, bias=True) # 은닉층2(30) -> 출력층(1)으로 가는 연산
        self.dropout = nn.Dropout(0.2) # 연산이 될 때 마다 20%의 비율로 랜덤하게 노드 없애기 overfitting 방지용!

    def forward(self, x): # 모델 연산의 순서 정의
        x = F.relu(self.fc1(x)) # Linear 계산 후 활성화 함수 ReLU 적용
        x = self.dropout(F.relu(self.fc2(x))) # 은닉층2에서 드롭아웃 적용 (즉, 30개에서 20%인 6개의 노드가 계산에서 제외)
        x = F.relu(self.fc3(x)) # Linear 계산 후 활성화 함수 ReLU 적용. 여기서 '집 값' 예측이기 때문에 값은 음수가 되면 안되므로 ReLU를 넣음.

        return x

# 주의!
# 드롭아웃은 overfitting을 방지하기 위해 노드의 일부를 배제하고 계산하는 것이기에 제대로!! 출력층에 사용해서선 안됨!
```



⇒ '집 값' 예측이기 때문에 값은 음수가 되면 안되므로 ReLU 함수 사용. 목적에 따라 다양한 활성화 함수 사용 가능

💡 활성화 함수 사용 이유: linear한 입력값을 non-linear하게 만들어 Linear 분류기의 한계 극복할 수 있음.

💡 다층신경망 MLP(Multiple layer perceptron)는 단지 linear layer를 여러개 쌓는 개념이 아닌 활성화 함수를 이용한 non-linear 시스템을 여러 layer로 쌓는 개념

[모델, 손실함수, 최적화 방법 선언]

```
model = Regressor()
criterion = nn.MSELoss()

# lr은 학습률
# weight_decay는 L2 정규화에서의 penalty 정도를 의미함.
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-7)
```

- **model.parameters()** 중요!

: optim.Adam()에는 어떤 기준으로 최적화를 할 것인가를 넣어주는 것

→ 우리는 모델의 **파라미터를 업데이트** 해야 하므로 model.parameters()를 기준으로 넣어줌

- 최적화 기법 | SGD, Adam, Adagrad 등 다양

5. 학습 진행

```
loss_ = [] # loss의 변화 그래프를 그리기 위한 loss 저장용 리스트
n = len(trainloader) # 배치의 개수

for epoch in range(400): # 400번 학습 진행
    running_loss = 0.0 # loss의 초기값은 0부터 시작해서 epoch가 둘 때마다 running loss에 loss를 더해주고 나중에 평균을 내줌

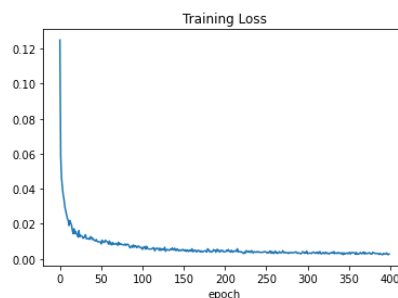
    for i, data in enumerate(trainloader, 0): # 무작위로 섞인 32개 데이터가 있는 배치가 하나씩 들어옴.
        inputs, values = data # data에는 X, Y가 들어있음.
        optimizer.zero_grad() # 최적화 초기화
        outputs = model(inputs) # 모델에 입력값 대입 후 예측값 산출
        loss = criterion(outputs, values) # 손실함수 계산 (MSE 계산)
        loss.backward() # 손실 함수 기준으로 역전파 설정
        optimizer.step() # 역전파를 진행하고 가중치를 업데이트

    running_loss += loss.item() # 매 epoch마다 평균 loss를 계산하기 위해 배치 loss를 더함.

    loss_.append(running_loss/n) # MSE

print('Finished Training')
```

```
plt.plot(loss_)
plt.title('Training Loss')
plt.xlabel('epoch')
plt.show()
```



→ overfitting이 있는 것 같음

6. 모델 평가 (testloader 이용)

```
def evaluation(data_loader):
    predictions = torch.tensor([], dtype=torch.float) # 예측값을 저장하는 텐서 생성
    actual = torch.tensor([], dtype=torch.float) # 실제값을 저장하는 텐서 생성

    with torch.no_grad(): # 최적화를 하지 않기 때문에 no_grad() 구간 하에 코드 작성(메모리 절약)
        model.eval() # 평가를 할 때는 .eval()를 반드시 사용! (중요) -> 드롭아웃 작동하지 않음!
        for data in data_loader:
            inputs, values = data
            outputs = model(inputs) # 드롭아웃이 비활성화된 결과값 출력

            predictions = torch.cat((predictions, outputs), 0) # cat을 통해 예측값 누적
            actual = torch.cat((actual, values), 0) # cat을 통해 실제값 누적

    predictions = predictions.numpy() # 넘파이 배열로 변경
    actual = actual.numpy()
    rmse = np.sqrt(mean_squared_error(predictions, actual)) # RMSE 계산
    return rmse
```

```
train_rmse = evaluation(train_loader)
test_rmse = evaluation(test_loader)

print(train_rmse, test_rmse)
```

💡 train_rmse : 0.041625913
test_rmse : 0.061637037

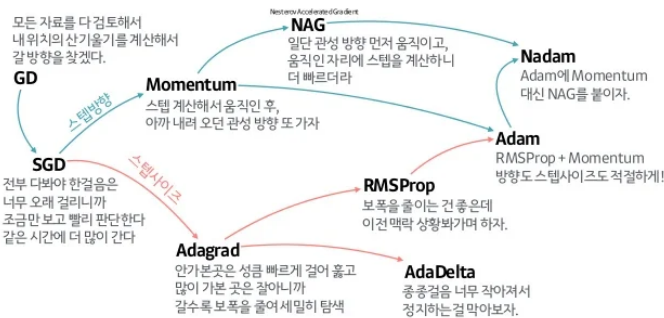
• 최적화 방법, 스케줄링, 교차검증

1. 다양한 최적화 기법

: 데이터의 크기가 클수록 훈련 속도가 느려지는 것은 상식.

⇒ 이 경우 효율성을 높이기 위해 최적화 알고리즘 사용

산내려오는 작은 오솔길 찾기기(Optimizer)의 발달 계보



```
import torch
import torchvision

# 모델 불러오기 (torchvision의 resnet18 모델)
model = torchvision.models.resnet18(pretrained=False)

# SGD 경사하강법
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Momentum (SGD에 momentum term만 추가해줌)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# Adam (요즘 거의 디폴트로 쓰는 것)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

2. 스케줄러

```
# Adam만큼 많이 쓰이는 스케줄링
# -> 최적화 기법(Adam 계열 제외)은 learning rate이 바뀌지 않기 때문에 스케줄링을 통해 learning rate을 강제로 바꿔준다
# 모멘텀 기반 최적화
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1)

# StepLR은 일정 스텝마다 학습률을 줄이거나 늘릴 수 있는 기능
# epoch이 30번 돌 때마다 learning rate에 gamma만큼 곱해서 learning rate 사용 (30-60 epoch때 learning rate은 0.01)
```


💡 이외에도 \cos 함수의 올라갔다 내려가는 성질을 이용해 learning rate를 늘렸다가 줄였다가 할 수 있는 CosineAnnealingLR도 있음.
-> local minimum에 빠졌을 때 다시 나올 수 있는 가능성 제시

3. 교차검증(Cross Validation)

: 데이터의 개수가 적은 경우, 학습 데이터를 쪼개어 일부는 학습 데이터로 사용하고, 나머지는 검증 데이터로 이용하는 것이 교차 검증!

💡 KFold cross validation은 학습 데이터를 k개의 조각으로 나누어 1개는 검증 데이터, 나머지 k-1개는 학습 데이터로 이용

```
# Cross Validation (교차검증)
from sklearn.model_selection import KFold

"""
나머지 모듈 및 클래스 중략
"""
# 전체 데이터를 학습 데이터와 평가 데이터로 나누자. (7:3)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.7)

trainset = TensorData(X_train, Y_train) # trainset을 이용하여 매 KFold마다 DataLoader를 만들 것
testset = TensorData(X_test, Y_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=32, shuffle=False)

"""
Regressor class 코드 중략
"""

kfold = KFold(n_splits=3, shuffle=True) # 학습 데이터를 몇개로 나눌 것인지 정함 (n_splits)
criterion = nn.MSELoss() # 목적함수

def evaluation(data_loader):
    predictions = torch.tensor([], dtype=torch.float) # 예측값을 저장하는 텐서 생성
    actual = torch.tensor([], dtype=torch.float) # 실제값을 저장하는 텐서 생성

    with torch.no_grad(): # 최적화를 하지 않기 때문에 no_grad() 구문 하에 코드 작성(메모리 절약)
        model.eval() # 평가를 할 때는 .eval()을 반드시 사용!(중요) -> 드롭아웃 작동하지 않음!
        """
        중략
        """
    rmse = np.sqrt(mean_squared_error(predictions, actual)) # RMSE 계산
    model.train()

    return rmse
```

💡 학습 중간에 평가 되기 때문에 위에서 model.eval()로 바꾼 것을 다시 model.train()으로 바꿔줘야 함!(중요)

→ 지금 예제에선 model.eval(), model.train() 둘 다 필요 없지만 모델에서 드롭아웃이나 배치정규화 등이 들어가면 두개 모두 필요함.

```
validation_loss = []

for fold, (train_idx, val_idx) in enumerate(kfold.split(trainset)):

    train_sub_sampler = torch.utils.data.SubsetRandomSampler(train_idx) # index 생성
    val_sub_sampler = torch.utils.data.SubsetRandomSampler(val_idx) # index 생성

    # sampler를 이용한 DataLoader 정의
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=32, sampler=train_sub_sampler) # index에 해당하는 2개를 trainloader에 저장
    valloader = torch.utils.data.DataLoader(trainset, batch_size=32, sampler=val_sub_sampler) # 나머지 1개 index에 해당하는 것을 valloader에 저장.

    # 모델
    model = Regressor()
    optimizer = optim.Adam(model.parameters(), lr=0.01, weight_decay=1e-7)

    for epoch in range(400):
        for data in trainloader:
            inputs, values = data
            optimizer.zero_grad() # 최적화 초기화
            outputs = model(inputs)

            loss = criterion(outputs, values) # 손실 함수 계산
            loss.backward()
            optimizer.step()

        train_rmse = evaluation(trainloader) # 학습 데이터 rmse
        val_rmse = evaluation(valloader)
        print("k-fold", fold, "Train Loss: %.4f, Validation Loss: %.4f" % (train_rmse, val_rmse))
        validation_loss.append(val_rmse)

validation_loss = np.array(validation_loss)
mean = np.mean(validation_loss)
std = np.std(validation_loss)
print("Validation Score: %.4f, ± %.4f" % (mean, std))
```

* BUT ! 딥러닝 분야에선 KFold 쓰기가 어려움 (안그래도 데이터 많아서 느린데 겹쳐서 하게 되면 더 오래걸림)

→ 그래서 처음부터 데이터를 나눌때 학습, 검증, 평가 set으로 나눔.

• 합성곱 신경망 (CNN)

- CIFAR10 데이터셋을 이용해 CNN을 이용한 이미지 분류를 해보자.



CNN은 이미지 처리에 특화된 모델

⇒ 다중채널로 여러개 이미지 feature를 계산하기 때문

```
# CIFAR10: 클래스 10개를 가진 이미지 데이터
# 'plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck' (10개의 클래스)

#전처리
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
# 컬러 이미지가 때문에 0,1,2채널에 대한 평균과 표준편차를 넣어줌.
```

[CNN 모델 구축]

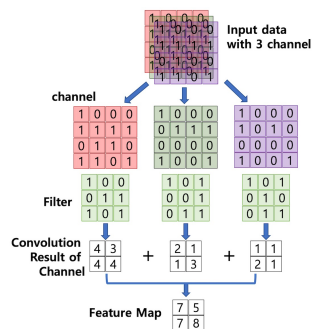
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5) # Convolutional 연산, 합성곱 연산 (입력 채널수 3, 출력 채널수 6, 필터크기 5x5, stride=1(default))
        self.pool1 = nn.MaxPool2d(2, 2) # MaxPooling 연산, 합성곱 연산 (필터크기 2, stride=2), 2x2 window를 이용해 maxpooling
        self.conv2 = nn.Conv2d(6, 16, 5) # 합성곱 연산 (입력 채널수 6, 출력 채널수 16, 필터크기 5x5, stride=1(default))
        self.pool2 = nn.MaxPool2d(2, 2) # 합성곱 연산 (필터크기 2, stride=2)
        self.fc1 = nn.Linear(16 * 5 * 5, 120) # 5x5 피쳐맵 16개를 일렬로 피면 16*5*5개의 노드가 생성, 120개의 노드를 가진 hidden layer 생성
        self.fc2 = nn.Linear(120, 10) # 120개 노드에서 클래스의 개수인 10개의 노드로 연산

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x))) # conv1 -> ReLU -> pool1
        x = self.pool2(F.relu(self.conv2(x))) # conv2 -> ReLU -> pool2
        x = x.view(-1, 16 * 5 * 5) # 5x5 피쳐맵 16개를 일렬로 만든다.
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x

net = Net().to(device) # 모델 선언, GPU 연산을 하기 위해 to(device)를 뒤에 적어야 함.
```

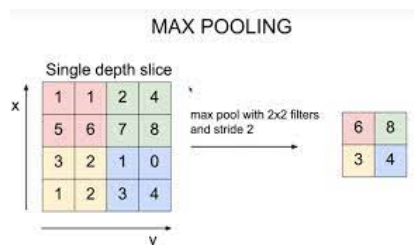
+Convolutional 연산



- 필터 크기 → 3x3

- stride → 1

+MaxPooling 연산



- 필터 크기 → 2x2

- stride → 2

좀 더 깔끔한 ver.

```
# nn.Sequential()을 이용해 좀 더 깔끔하게 CNN 모델 구축
class Net(nn.Module):
    def __init__(self):
```

```

super(Net, self).__init__()

self.feature_extraction = nn.Sequential(nn.Conv2d(3, 6, 5),
                                       nn.ReLU(),
                                       nn.MaxPool2d(2, 2),
                                       nn.Conv2d(6, 16, 5),
                                       nn.ReLU(),
                                       nn.MaxPool2d(2, 2)) # 차례대로 연산

self.classifier = nn.Sequential(nn.Linear(16 * 5 * 5, 120),
                                nn.ReLU(),
                                nn.Linear(120, 10)) #fully connected layer

def forward(self, x):
    x = self.feature_extraction(x)
    x = x.view(-1, 16 * 5 * 5) # 5x5 피쳐맵 16개를 일렬로 만든다.
    x = self.classifier(x)

    return x

net = Net().to(device) # 모델 선언

```

- 순환 신경망 (RNN)
- LSTM , GRU
- BI - LSTM

Section 04 | Pretrained Model

[학습된 모델 | Pretrained Model]

: 전이학습이란 기존의 잘 알려진 데이터 혹은 사전학습 된 모델을 업무 효율 증대나 도메인 확장을 위해 사용하는 학습을 의미한다. 따라서 전이학습은 인공지능 분야에서 매우 중요한 연구 중 하나이며 다양한 방법론들이 존재한다. 8강에서는 잘 학습 된 모델을 재사용하는 방법에 대해서 설명한다.

- 라이브러리 & GPU 연산

```

import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

import torch.nn as nn
import torch.optim as optim

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

```

- CIFAR10 데이터 불러오기 & 전처리

```

# 데이터 불러오기 및 전처리 작업
transform = transforms.Compose(
    [transforms.RandomCrop(32, padding=4),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

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

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

trainloader = torch.utils.data.DataLoader(trainset, batch_size=16, shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=test_transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=16, shuffle=False)

```

: 전처리 하는 `tr.Compose` 안에 `RandomCrop` 함수는 원래는 (40,40)의 이미지를 (32,32)로 랜덤하게 자르는 함수로 data augmentation에 이용한다.

: 데이터 로더의 배치 사이즈는 16이다.

- Pretrained Model

: 이미 누군가가 학습한 모델을 불러와서 사용한다. pytorch에서는 여러 pretrained model을 제공하고 있다. 불러와서 사용하기만 하면 된다!

- 불러오기

```

# ResNet18 불러오기
# pretrained=True를 하면 ResNet18 구조와 사전 학습 된 파라메터를 모두 불러온다.
# pretrained=False를 하면 ResNet18 구조만 불러온다.
# 모델과 텐서에 .to(device)를 붙여야만 GPU 연산이 가능하니 꼭 기입한다.

model = torchvision.models.resnet18(pretrained=True)

```

: 이 모델의 구조를 살펴 보면, 마지막 fc layer에 out_features=1000 으로 나와 있다. 그러나 우리가 쓸 데이터는 10개 class 로의 분류되기 때문에 이 부분을 바꿔 주면 된다.

◦ 출력 노드 바꾸기

```
# 모델의 구조를 보면 마지막 출력 노드가 1000개라는 것을 알 수 있다.
# 이는 1000개의 클래스를 가진 ImageNet 데이터를 이용하여 사전학습 된 모델이기 때문이다.
# 따라서 우리가 사용하는 CIFAR10 데이터에 맞게 출력층의 노드를 10개로 변경해야만 한다.

num_ftrs = model.fc.in_features # fc의 입력 노드 수를 산출한다. 512개
model.fc = nn.Linear(num_ftrs, 10) # fc를 nn.Linear(num_ftrs, 10)로 대체한다.
model = model.to(device)
```

◦ 손실함수와 최적화 방법 정의

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-2)
```

• 학습하기

```
for epoch in range(20):

    running_loss = 0.0
    for data in trainloader:

        inputs, labels = data[0].to(device), data[1].to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    cost = running_loss / len(trainloader)
    print('[%d] loss: %.3f' %(epoch + 1, cost))

torch.save(model.state_dict(), './models/cifar10_resnet18.pth')

print('Finished Training')
```

• 모델 평가

```
model = torchvision.models.resnet18(pretrained=False)
num_ftrs = model.fc.in_features # fc의 입력 노드 수를 산출한다. 512개
model.fc = nn.Linear(num_ftrs, 10) # fc를 nn.Linear(num_ftrs, 10)로 대체한다.
model = model.to(device)
# 모델 학습 후 저장된 모델을 다시 불러올 때 필요한 과정
# 모델에 parameter 를 새로 넣는 것이기 때문에 빈 모델 필요

model.load_state_dict(torch.load('./models/cifar10_resnet18.pth'))
```

```
correct = 0
total = 0
with torch.no_grad():
    model.eval()
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (100 * correct / total))
```

◦ 모델 성능의 향상 요인

- (1) 더 깊은 모델
- (2) random crop을 통한 data augmentation
- (3) pretrained model

⇒ pretrained model을 쓰면 random weight로 시작하는 것이 아닌 유사한 task로 학습되어 있는 weight로 시작하기 때문에 최적화된 weight에 도달하기 더욱 쉽다!

[Model Freezing | 모델 프리징]

: 전이 학습 중 잘 학습 된 모델을 가져와 우리 연구에 사용할 수 있다. 데이터가 유사한 경우에는 추가적인 전체 학습 없이도 좋은 성능이 나올 수 있다. 따라서 **피쳐(feature) 추출에 해당하는 합성곱 층의 변수를 업데이트 하지 않고 분류 파트에 해당하는 fully connected layer 의 변수만**

업데이트 할 수 있는데 이 때 변수가 업데이트 되지 않게 변수를 얼린다고 하여 이를 프리징(Freezing)이라고 한다.

⇒ 즉, requires_grad를 사용하지 않는것

• 사전 준비

- 라이브러리 호출

```
import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

import torch.nn as nn
import torch.optim as optim
```

- GPU 연산 확인 및 데이터 불러오기

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

# 데이터 불러오기 및 전처리 작업
transform = transforms.Compose(
    [transforms.RandomCrop(32, padding=4),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

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

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

trainloader = torch.utils.data.DataLoader(trainset, batch_size=16, shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=test_transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=16, shuffle=False)
```

⇒ random crop 을 통한 data augmentation 진행

• Pretrained Model 불러오기

: 이번에는 AlexNet을 불러온다!

→ AlexNet도 ResNet과 마찬가지로 ImageNet으로 학습되었기 때문에 output node가 1000개인데, 10개로 바꿔준다.

```
model = torchvision.models.alexnet(pretrained=True)

num_fts = model.classifier[6].in_features # fc의 입력 노드 수를 산출한다.
model.classifier[6] = nn.Linear(num_fts, 10) # fc를 nn.Linear(num_fts, 10)로 대체한다.
model = model.to(device)
```

• 모델 프리징

- 파라미터 번호 확인

```
# model.named_parameters() 이용
i = 0
for name, param in model.named_parameters():

    print(i, name) # 일단 name만 사용
    i += 1
```

: 이렇게 하면 가중치가 있는 레이어의 번호들을 알 수 있다. 결과를 보면 0~9까지는 features의 parameter, 10~15까지가 classifier의 parameter 이기 때문에 0~9까지의 parameters 만 requires_grad 를 False 로 변경

- 0~9까지의 requires_grad = False

```
# 합성곱 층은 0~9까지이다. 따라서 9번째 변수까지 역추적을 비활성화 한 후 for문을 종료한다.

for i, (name, param) in enumerate(model.named_parameters()):

    param.requires_grad = False
    if i == 9:
        print('end')
        break
```

⇒ freezing

- 이후 동일하게 손실함수 최적화, 학습, 평가 순서로 진행

: 이때 아무리 학습 해도 CNN 연산의 weight는 초기에 불러온 pretrained weight를 가짐.

Section 05 | GAN & K-Means Clustering

[AutoEncoder | 오토 인코더]

: 오토인코더는 라벨 없이 원하는 출력값을 생성하는 모델이다. 따라서 **모델의 파라메타들을 학습하기 위해 라벨을 사용하지 않는 비지도 학습 (unsupervised learning)**에 속한다. 오토인코더는 많은 연구에 기반이 되는 기본적인 비지도 학습 모델 중 하나다.

- 라이브러리 호출 및 연산 타입 확인

```
import torch
import torchvision
from torchvision import transforms
import torch.nn.functional as F

import torch.nn as nn
import torch.optim as optim

import numpy as np
import cv2
import matplotlib.pyplot as plt

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f'{device} is available.')
```

- MNIST Data

```
dataset = torchvision.datasets.MNIST('./data/', download=True, train=True, transform=transforms.ToTensor())
trainloader = torch.utils.data.DataLoader(dataset, batch_size=50, shuffle=True)
```

- 오토 인코더 모델 생성

: 오토 인코더는 인코더/디코더 단으로 나뉘

→ AutoEncoder 라는 하나의 class 안에서 Encoder 부분과 Decoder 부분으로 분리

```
class Flatten(torch.nn.Module): # 4D -> 2D로 계산하기
    def forward(self, x):
        batch_size = x.shape[0]
        return x.view(batch_size, -1) # (배치 수, 채널 수, 이미지 너비, 이미지 높이) -> (배치 수, 채널 수*이미지 너비*이미지 높이)

class DeFlatten(torch.nn.Module): # 2D -> 4D로 계산하기

    def __init__(self, k):
        super(DeFlatten, self).__init__()
        self.k = k

    def forward(self, x):
        s = x.size()

        # 벡터 사이즈 = 채널 수*이미지 너비*이미지 높이
        # 벡터 사이즈 = 채널 수*이미지 사이즈**2
        # 이미지 사이즈 = (벡터 사이즈/채널 수)**.5
        feature_size = int((s[1]/self.k)**.5)

        return x.view(s[0], self.k, feature_size, feature_size) # (배치 수, 채널 수*이미지 너비*이미지 높이) -> (배치 수, 채널 수, 이미지 너비, 이미지 높이)

class Autoencoder(torch.nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()

        k = 16
        self.encoder = nn.Sequential(
            nn.Conv2d(1, k, 3, stride=2),
            nn.ReLU(),
            nn.Conv2d(k, 2*k, 3, stride=2),
            nn.ReLU(),
            nn.Conv2d(2*k, 4*k, 3, stride=1),
            nn.ReLU(),
            Flatten(),
            nn.Linear(1024, 10),
            nn.ReLU()
        )
        # padding이 없기 때문에 size가 계속 줄어든다.

        # ConvTranspose2d
        # 입력 성분(Conv의 결과)을 출력 성분(Conv의 입력)으로 미분하여 그 값을 입력 벡터와 곱해 출력 벡터를 산출한다.
        # 출력 된 벡터는 행렬 형태로 변환한다.
        self.decoder = nn.Sequential(
            nn.Linear(10, 1024),
            nn.ReLU(),
            DeFlatten(4*k),
            nn.ConvTranspose2d(4*k, 2*k, 3, stride=1), # (입력 채널 수, 출력 채널 수, 필터 크기, stride)
            nn.ReLU(),
            nn.ConvTranspose2d(2*k, k, 3, stride=2),
            nn.ReLU(),
            nn.ConvTranspose2d(k, 1, 3, stride=2, output_padding=1),
            nn.Sigmoid()
        )
```

```
)

def forward(self, x):

    encoded = self.encoder(x)
    decoded = self.decoder(encoded)

    return decoded
```

→ decoder 단에서 사이즈를 늘릴 때 그냥 convolution을 쓸 수 없고 `ConvTranspose` 라는 함수를 이용한다.

- `ConvTranspose` 는 Upsampling에 쓰는 함수이다.

$$O = (I - 1)S - 2P_{input} + K + P_{output}$$

O = output size, I = input size, S = stride, P_{input} = input padding, P_{output} = output padding, K = kernel size

```
model = Autoencoder().to(device)
```

• 시각화 함수

: 모델이 학습 하면서 나오는 결과를 시각화 하기 위함

```
def normalize_output(img):
    img = (img - img.min())/(img.max()-img.min())
    return img
```

```
def check_plot():
    with torch.no_grad():
        for data in trainloader:

            inputs = data[0].to(device)
            outputs = model(inputs)

            input_samples = inputs.permute(0,2,3,1).cpu().numpy() # 원래 이미지
            reconstructed_samples = outputs.permute(0,2,3,1).cpu().numpy() # 생성 이미지
            break # 배치 하나만 받고 for문 종료

            #reconstructed_samples = normalize_output(reconstructed_samples) # 0-1사이로 변환
            #input_samples = normalize_output(input_samples) # 0-1사이로 변환

            columns = 10 # 시각화 전체 너비
            rows = 5 # 시각화 전체 높이

            fig=plt.figure(figsize=(columns, rows)) # figure 선언

            # 원래 이미지 배치 크기 만큼 보여주기
            for i in range(1, columns*rows+1):
                img = input_samples[i-1]
                fig.add_subplot(rows, columns, i)
                plt.imshow(img.2) # 1채널인 경우 2로 변환
                #plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
                plt.axis('off')
            plt.show()

            # 생성 이미지 배치 크기 만큼 보여주기
            fig=plt.figure(figsize=(columns, rows))

            for i in range(1, columns*rows+1):
                img = reconstructed_samples[i-1]
                fig.add_subplot(rows, columns, i)
                plt.imshow(img.squeeze())
                #plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
                plt.axis('off')
            plt.show()
```

→ 예시이기 때문에 전체적으로 학습이 되지 않고 한 번만 돌고 끝남.(break) 그 이하는 subplot

• 오토 인코더 학습

- 손실 함수 및 최적화 방법 정의

```
criterion = nn.MSELoss() # MSE 사용
optimizer = optim.Adam(model.parameters(), lr=1e-4)
```

: 입력값과 출력값의 차이를 loss로 이용함

- 학습

```
for epoch in range(51):

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):

        inputs = data[0].to(device) # 여기에 label이 없음
```

```
optimizer.zero_grad()
outputs = model(inputs)

loss = criterion(inputs, outputs) # 라벨 대신 입력 이미지와 출력 이미지를 비교

loss.backward()
optimizer.step()
running_loss += loss.item()

cost = running_loss / len(trainloader)

if epoch % 10 == 0:
    print('[%d] loss: %.3f' %(epoch + 1, cost))
    check_plot()
```



[GAN | 생성적 적대 신경망]

: 생성적 적대 신경망은 아이디어 자체로만으로도 매우 가치있는 모델로 평가 받고 있다. 실제 얼굴 변환, 생성, 음성 변조, 그림 스타일 변환, 사진 복원 등 다양한 기술로 응용이 되어 실제 적용되고 있다.

• 라이브러리

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.autograd.variable import Variable
from torchvision import transforms
from torchvision.datasets import FashionMNIST
from torchvision.utils import make_grid
from torch.utils.data import DataLoader
import imageio

import numpy as np
from matplotlib import pyplot as plt
```

• 데이터 | FashionMNIST

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,),(0.5,))
])
to_image = transforms.ToPILImage()
trainset = FashionMNIST(root='./data/', train=True, download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=100, shuffle=True)

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

# 'T-Shirt','Trouser','Pullover','Dress','Coat','Sandal','Shirt','Sneaker','Bag','Ankle Boot'
```

• 모델 생성

◦ 이미지를 생성하는 Generator

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.n_features = 128 # latent space에서 나가는 vector의 크기
        self.n_out = 784 # Fashion MNIST가 28x28이어서
        self.linear = nn.Sequential(
            nn.Linear(self.n_features, 256),
            nn.LeakyReLU(0.2), # 기울기
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, self.n_out),
            nn.Tanh()
        )
        # 이번엔 바닐라 Generator라 Linear
```



```
def forward(self, x):
    x = self.linear(x)
    x = x.view(-1, 1, 28, 28)
    return x
```

: 이미지를 생성해야 하기 때문에 deconvolution 이 필요함. (오늘은 Vanila라서 Linear)

→ Loss가 Min-max 라서 두 번 업데이트 해야 하는데, 업데이트가 잘 안 되는 경우가 있음
(LeakyReLU 사용)

◦ 진짜/가짜 이미지를 판별하는 Discriminator

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.n_in = 784
        self.n_out = 1
        self.linear = nn.Sequential(
            nn.Linear(self.n_in, 1024),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(256, self.n_out),
            nn.Sigmoid() # T or F를 구분
        )
    def forward(self, x):
        x = x.view(-1, 784)
        x = self.linear(x)
        return x
```

→ 28x28 이 다시 784 의 벡터가 되어서 들어감

• 손실 함수 및 최적화 방법 정의

◦ 모델 불러오기

```
generator = Generator().to(device)
discriminator = Discriminator().to(device)
```

◦ pretrained가 있는 경우

```
pretrained = False # pretrained 있으면 쉼
if pretrained == True:
    discriminator.load_state_dict(torch.load('./models/fmnist_disc.pth'))
    generator.load_state_dict(torch.load('./models/fmnist_gner.pth'))
```

◦ Optimizer 설정

```
g_optim = optim.Adam(generator.parameters(), lr=2e-4)
d_optim = optim.Adam(discriminator.parameters(), lr=2e-4)
```

→ 두 단으로 나뉘기 때문에 optimizer 도 두개!

◦ 정의

```
g_losses = []
d_losses = []
images = [] # 음뽕처럼 만들어보려고 images list

criterion = nn.BCELoss() # Binary Cross Entropy

def noise(n, n_features=128):
    return Variable(torch.randn(n, n_features)).to(device)

def label_ones(size):
    data = Variable(torch.ones(size, 1))
    return data.to(device)

def label_zeros(size):
    data = Variable(torch.zeros(size, 1))
    return data.to(device)
```

- noise : latent space의 vector. 즉, Generator로 들어오는 input값. 아무 input값 (128의 vector)를 랜덤으로 만들어서 generator에 주면 generator가 28x28짜리 image를 만들어내는 것
- label_ones : 진짜일 때 1, label_zeros : 가짜일 때

• 학습 전략 정의

◦ Discriminator 의 학습

```
def train_discriminator(optimizer, real_data, fake_data):
    n = real_data.size(0)

    optimizer.zero_grad()

    prediction_real = discriminator(real_data)
    d_loss = criterion(prediction_real, label_ones(n)) # 진짜 기준

    prediction_fake = discriminator(fake_data)
    g_loss = criterion(prediction_fake, label_zeros(n)) # 가짜 기준

    loss = d_loss + g_loss

    loss.backward()
    optimizer.step()

    return loss.item()
```

: 진짜 이미지 기준으로 한 번, 가짜 이미지로 한 번 각각 loss를 구해서 더한 것이 discriminator에서의 loss

◦ Generator의 학습

```
def train_generator(optimizer, fake_data):
    n = fake_data.size(0)
    optimizer.zero_grad()

    prediction = discriminator(fake_data)
    loss = criterion(prediction, label_ones(n))

    loss.backward()
    optimizer.step()

    return loss.item()
```

⇒ Discriminator 가 분류를 못 하게 만드는 것이 목표

: fake image + 진짜 label을 넣었을 때의 loss를 계산함. 이 loss가 작으면 구분을 못 하고 크면 구분을 잘한다는 뜻

• 학습하기

```
num_epochs = 201
test_noise = noise(64) # generator로 들어가는 벡터 (128의 벡터를 64개 생성)

l = len(trainloader)

for epoch in range(num_epochs):
    g_loss = 0.0
    d_loss = 0.0

    for data in trainloader:
        imgs, _ = data # 비지도 학습이기 때문에 label을 불러오지 않음
        n = len(imgs)

        fake_data = generator(noise(n)).detach() # input (noise)이 들어옴
        real_data = imgs.to(device) # 진짜 이미지는 real_data라고 정의
        d_loss += train_discriminator(d_optim, real_data, fake_data) # discriminator에 진짜와 가짜 데이터 모두 이미지 넣고 discriminator만 업데이트 할 거라서 파라미터로 d_optim
        fake_data = generator(noise(n))
        g_loss += train_generator(g_optim, fake_data) # g_optim에 대해서만 최적화

    img = generator(test_noise).cpu().detach()
    img = make_grid(img)
    images.append(img)
    g_losses.append(g_loss/l)
    d_losses.append(d_loss/l)

    if epoch % 10 == 0:
        print('Epoch {}: g_loss: {:.3f} d_loss: {:.3f}\r'.format(epoch, g_loss/l, d_loss/l))

torch.save(discriminator.state_dict(), './models/fmnist_disc.pth')
torch.save(generator.state_dict(), './models/fmnist_gner.pth')
print('Training Finished')
```

[Deep-K Means | AutoEncoder + K-Means]

▼ 특징

- Auto Encoder 는 encoder 부분에서 정보가 압축 → latent variable 이 되고, Decoder단을 거치며 유사한 데이터를 만드는 것
- latent variable은 이미지에 대한 중요한 정보를 담고 있을 것임!!

- latent variable을 이용해서 이미지를 분류 한다면 비지도 학습으로서 좋은 결과를 얻지 않을까?

▼ 장점

- 차원 축소 가능
- 비지도 학습 가능
- latent variable을 가지고 k-means에 넣으면 됨

• 라이브러리

```
import torch
import torch.nn as nn
import torchvision
from torchvision import transforms
import numpy as np
from scipy.optimize import linear_sum_assignment as linear_assignment # clustering 평가를 위해서
from sklearn.manifold import TSNE # 고차원 시각화
from matplotlib import pyplot as plt
```

• 데이터 불러오기

```
batch_size = 128
num_clusters = 10
latent_size = 10 # cluster 개수와 같은 필요 x 임의로 정하면 된다.
```

```
trainset = torchvision.datasets.MNIST('./data/', download=True, train=True, transform=transforms.ToTensor())
testset = torchvision.datasets.MNIST('./data/', download=True, train=False, transform=transforms.ToTensor())

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

• 모델 정의

◦ K-Means 정의

```
class Flatten(torch.nn.Module):
    def forward(self, x):
        batch_size = x.shape[0]
        return x.view(batch_size, -1)

class Deflatten(nn.Module):
    def __init__(self, k):
        super(Deflatten, self).__init__()
        self.k = k

    def forward(self, x):
        s = x.size()
        feature_size = int((s[1]//self.k)**.5)
        return x.view(s[0], self.k, feature_size, feature_size)

class Kmeans(nn.Module):
    def __init__(self, num_clusters, latent_size):
        super(Kmeans, self).__init__()
        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        self.num_clusters = num_clusters
        self.centroids = nn.Parameter(torch.rand((self.num_clusters, latent_size)).to(device))
        # 중심을 어떻게 정하느냐가 중요하기 때문에 centroid가 parameter

    def argminl2distance(self, a, b):
        return torch.argmax(torch.sum((a-b)**2, dim=1), dim=0) # centroid에 대한 거리 계산

    def forward(self, x):
        y_assign = []
        for m in range(x.size(0)):
            h = x[m].expand(self.num_clusters, -1)
            assign = self.argminl2distance(h, self.centroids)
            y_assign.append(assign.item())

        return y_assign, self.centroids[y_assign]
```

◦ AutoEncoder

```
class Encoder(nn.Module):
    def __init__(self, latent_size):
        super(Encoder, self).__init__()

        k = 16
        self.encoder = nn.Sequential(
            nn.Conv2d(1, k, 3, stride=2),
            nn.ReLU(),
            nn.Conv2d(k, 2*k, 3, stride=2),
            nn.ReLU(),
            nn.Conv2d(2*k, 4*k, 3, stride=1),
            nn.ReLU(),
```

```

        Flatten(),
        nn.Linear(1024, latent_size),
        nn.ReLU()
    )

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

class Decoder(nn.Module):
    def __init__(self, latent_size):
        super(Decoder, self).__init__()

        k = 16
        self.decoder = nn.Sequential(
            nn.Linear(latent_size, 1024),
            nn.ReLU(),
            Deflatten(4*k),
            nn.ConvTranspose2d(4*k, 2*k, 3, stride=1), # (입력 채널 수, 출력 채널 수, 필터 크기, stride)
            nn.ReLU(),
            nn.ConvTranspose2d(2*k, k, 3, stride=2),
            nn.ReLU(),
            nn.ConvTranspose2d(k, 1, 3, stride=2, output_padding=1),
            nn.Sigmoid()
        )

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

```

• 클러스터링 정확도 함수 정의

```

def cluster_acc(y_true, y_pred):

    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    D = max(y_pred.max(), y_true.max()) + 1
    w = np.zeros((D, D), dtype=np.int64)
    for i in range(y_pred.size):
        w[y_pred[i], y_true[i]] += 1
    ind = linear_assignment(w.max() - w)

    return sum([w[i, j] for i, j in zip(ind[0], ind[1])]) * 1.0 / y_pred.size

def evaluation(testloader, encoder, kmeans, device):
    predictions = []
    actual = []

    with torch.no_grad():
        for images, labels in testloader:
            inputs = images.to(device)
            labels = labels.to(device)
            latent_var = encoder(inputs)
            y_pred, _ = kmeans(latent_var)

            predictions += y_pred
            actual += labels.cpu().tolist()

    return cluster_acc(actual, predictions)

```

: 클러스터링 시 결과가 label이 0인건 0번째 클러스터로, 1은 1번째 클러스터로 순차적으로 가지 않음 → label이 0, 5, 7인게 0번째 클러스터 label이 1, 2, 3인게 1번째 클러스터로 감

⇒ accuracy 평가 할 때 재조정 필요

- confusion matrix 처럼 10 by 10 matrix 를 만든 후, 각각의 F/T 로 값을 하나씩 더함 ex] (True, pred) = (0,0) 이면 w matrix의 [0,0] 에 1 누적

• 손실 함수 및 최적화 방법 정의

```

encoder = Encoder(latent_size).to(device)
decoder = Decoder(latent_size).to(device)
kmeans = Kmeans(num_clusters, latent_size).to(device)

```

```

# Loss and optimizer
criterion1 = torch.nn.MSELoss() # AutoEncoder Loss
criterion2 = torch.nn.MSELoss() # Centroid of Cluster의 loss
optimizer = torch.optim.Adam(list(encoder.parameters()) +
                               list(decoder.parameters()) +
                               list(kmeans.parameters()), lr=1e-3)

```

: Loss가 두 개로 운영됨. Optimizer에 parameter를 여러개 넣고 싶을 땐 list의 합으로 넣기

• 학습하기

```

# Training
T1 = 50
T2 = 200
lam = 1e-3
ls = 0.05

```

```

for ep in range(300):
    if (ep > T1) and (ep < T2):
        alpha = lam*(ep - T1)/(T2 - T1) # 1/100, 2/100, ..., 99/100
    elif ep >= T2:
        alpha = lam
    else:
        alpha = lam/(T2 - T1)

    running_loss = 0.0

    for images, _ in trainloader:
        inputs = images.to(device)
        optimizer.zero_grad()
        latent_var = encoder(inputs)
        _, centroids = kmeans(latent_var.detach())
        outputs = decoder(latent_var)

        l_rec = criterion1(inputs, outputs) # Auto Encoder Loss
        l_clt = criterion2(latent_var, centroids) # Cluster Loss
        loss = l_rec + alpha*l_clt
        #동시에 같은 크기로 optimizing되면 autoencoder에 해당하는 변수들이 최적화가 잘 안 되는 경우가 있어서 alpha를 곱함 (alpha의 값은 작게-조금씩 증가하게)
        loss.backward()
        optimizer.step()

    running_loss += loss.item()

avg_loss = running_loss / len(trainloader)

if ep % 10 == 0:
    testacc = evaluation(testloader, encoder, kmeans, device)
    print('[%d] Train loss: %.4f, Test Accuracy: %.3f' %(ep, avg_loss, testacc))

if avg_loss < ls:
    ls = avg_loss
    torch.save(encoder.state_dict(), './models/dkm_en.pth')
    torch.save(decoder.state_dict(), './models/dkm_de.pth')
    torch.save(kmeans.state_dict(), './models/dkm_clt.pth')

```

Section 06 | Visualization

[CAM]

: CAM(Class Activation Map)은 설명 가능한 AI 기술 중 하나다. 이미지 내에서 어느 부분이 모델의 의사결정에 큰 영향을 미쳤는지를 시각화 하는 기술이다.

- 라이브러리 호출 및 구글드라이브 연동

```

import numpy as np
from matplotlib import pyplot as plt
import cv2

import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim

```

- 데이터 및 모델 생성

```

# CIFAR10보다 size가 큰 STL10 이미지를 사용
# 10 classes: airplane, bird, car, cat, deer, dog, horse, monkey, ship, truck

# 96x96 -> 128 X 128
transform = transforms.Compose([transforms.Resize(128), transforms.ToTensor()])
trainset = torchvision.datasets.STL10(root='./data', split='train',
                                     download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=40, shuffle=True)

model = torchvision.models.resnet18(pretrained=True)
num_fts = model.fc.in_features # fc의 입력 노드 수를 산출한다. 512개

# fc를 nn.Linear(num_fts, 10)로 대체한다. / 마지막 CLASS의 개수를 10개로 맞춰줘야 함.
model.fc = nn.Linear(num_fts, 10)
model = model.to(device) ## GPU 연산 가능하게

```

- 모델 학습

```

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-2)

for epoch in range(20):

    running_loss = 0.0
    for data in trainloader:

```

```

inputs, labels = data[0].to(device), data[1].to(device)

optimizer.zero_grad()
outputs = model(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()

running_loss += loss.item()

cost = running_loss / len(trainloader)
print('[%d] loss: %.3f' %(epoch + 1, cost))

torch.save(model.state_dict(), './models/stl10_resnet18.pth')

print('Finished Training')

correct = 0
total = 0
with torch.no_grad():
    model.eval()
    for data in trainloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the train images: %d %%' % (100 * correct / total))

## Accuracy of the network on the train images: 99 %

```

• 모델 구조 (일부)

```

ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)

  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  .
  .
  .

  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)

## CAM은 feature_extraction의 마지막 feature map을 기준으로 시각화
## RESNET에서 avgpooling 직전이 마지막 feature map!
## 그래서 해당 정보를 bn2에서 뽑은 후,
## fc에서 각 feature map의 가중치를 얻어 bc2와 fc를 곱연산

```

• CAM 모델 구축

```

# Visualize feature maps ( 모델 내에서 filter나 feature map을 뽑아낼 때 사용)

activation = {}

```

```
def get_activation(name):
    def hook(model, input, output):
        activation[name] = output.detach()
    return hook
```

```
def cam(model, trainset, img_sample, img_size):
    model.eval() ## batch_norm이 학습할 때의 구동 방법으로 구동이 되는 것을 방지하기 위해
    with torch.no_grad(): # requires_grad 비활성화

        # feature extraction 위한 마지막 feature map 구하기
        model.layer4[1].bn2.register_forward_hook(get_activation('final'))
        data, label = trainset[img_sample] # 이미지 한 장과 라벨 불러오기

        data.unsqueeze_(0) # 3차원 -> 4차원 [피쳐수, 너비, 높이] -> [1, 피쳐수, 너비, 높이]
        output = model(data.to(device))
        _, prediction = torch.max(output, 1)
        act = activation['final'].squeeze() # 4차원 [1, 피쳐수, 너비, 높이] -> 3차원 [피쳐수, 너비, 높이]

        w = model.fc.weight # classifier의 가중치 불러오기

        for idx in range(act.size(0)): # CAM 연산
            if idx == 0:
                tmp = act[idx] * w[prediction.item()][idx]
            else:
                tmp += act[idx] * w[prediction.item()][idx]

        # normalize 후 모든 이미지 픽셀값을 0~255로 스케일하기
        original_img = np.uint8((data[0][0] / 2 + 0.5) * 255)

        normalized_cam = tmp.cpu().numpy()
        normalized_cam = (normalized_cam - np.min(normalized_cam)) / (np.max(normalized_cam) - np.min(normalized_cam))

        # 원본 이미지 사이즈로 리사이즈
        cam_img = cv2.resize(np.uint8(normalized_cam * 255), dsize=(img_size, img_size))

    return cam_img, original_img

def plot_cam(model, trainset, img_size, start):
    end = start + 20 ## 결과 값 20개가 나오게
    fig, axs = plt.subplots(2, (end - start + 1) // 2, figsize=(20, 5))
    fig.subplots_adjust(hspace=.01, wspace=.01)
    axs = axs.ravel()

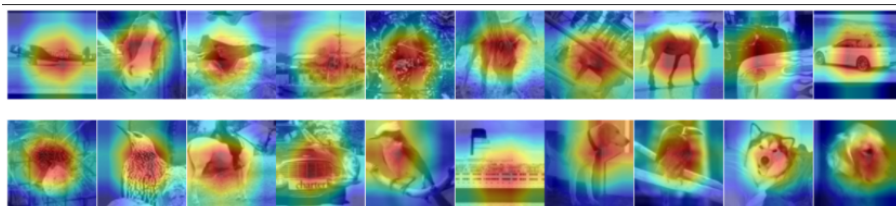
    for i in range(start, end):
        cam_img, original_img = cam(model, trainset, i, img_size)

        axs[i - start].imshow(original_img, cmap='gray')
        axs[i - start].imshow(cam_img, cmap='jet', alpha=.5)
        axs[i - start].axis('off')

    plt.show()
    fig.savefig('cam.png')

## 실행
plot_cam(model, trainset, 128, 10)
```

• 결과



[결과 시각화]

• t-SNE(t-distributed Stochastic Neighbor Embedding)

: 실험이 완료되면 결과를 시각화 하는 것이 매우 중요하다. 하지만 이미지나 추출 된 피쳐들은 차원이 높기 때문에 각 피쳐들이 어떤 분포를 가지고 있는지 표현하기 어렵다. 다시 말해서, 우리가 시각적으로 표현할 수 있는 차원은 3차원이기 때문에 고차원의 벡터들을 3차원 이하의 저차원으로 바꿔야만 한다. 따라서 이를 위해 널리 쓰이는 방법 중 하나인 t-SNE를 구현해 본다.

• 라이브러리

```
from sklearn.manifold import TSNE
import numpy as np
from matplotlib import pyplot as plt

import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn as nn
```

- 데이터 불러오기

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=16)
```

- 모델 불러오기

```
class Identity(nn.Module):
    def __init__(self):
        super(Identity, self).__init__()

    def forward(self, x):
        return x

model = torchvision.models.resnet18(pretrained=False)
num_fts = model.fc.in_features # fc의 입력 노드 수를 산출한다. 512개
model.fc = nn.Linear(num_fts, 10) # fc를 nn.Linear(num_fts, 10)로 대체한다.
model = model.to(device)

# 목적 : 합성곱 층의 마지막 단계에 있는 feature map으로
# 각각의 10개의 class들의 분포를 알아보자!
# 그래서, classifier에 들어가기 직전의 avgpool 값을 뽑아 사용해보자!
model.load_state_dict(torch.load('./models/cifar10_resnet18.pth'))
model.fc = Identity() ## 512개의 값을 그대로 내보내는 것.
```

```
[ 모델 구조 ]
(layer4): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(fc): Identity()
```

- t-SNE

```
actual = []
deep_features = []

model.eval()
with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        features = model(images)

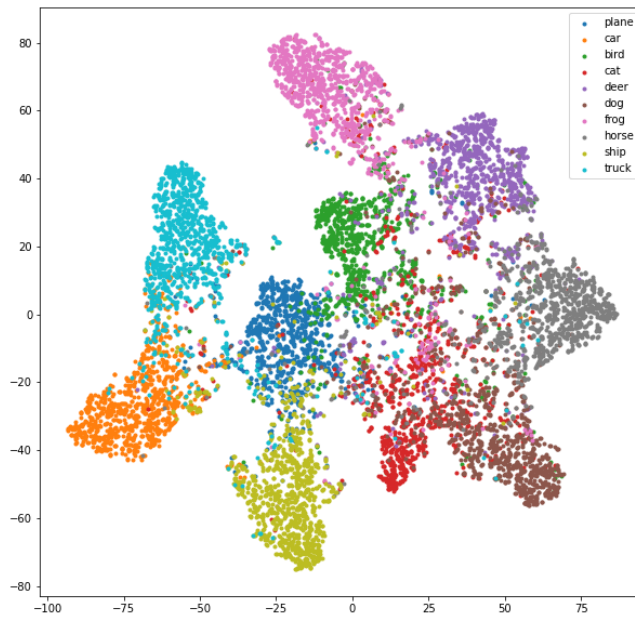
        deep_features += features.cpu().numpy().tolist()
        actual += labels.cpu().numpy().tolist()

tsne = TSNE(n_components=2, random_state=0) ## 어떠한 점을 기준으로 잡는지에 대한 seed 설정
cluster = np.array(tsne.fit_transform(np.array(deep_features)))
actual = np.array(actual)

plt.figure(figsize=(10, 10))
cifar = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
for i, label in zip(range(10), cifar):
    idx = np.where(actual == i)
    plt.scatter(cluster[idx, 0], cluster[idx, 1], marker='.', label=label)

plt.legend()
plt.show()

# 해석 : plane , bird 비스 / car, truck 비스 / cat, dog 비스.. / horse, truck 다음.. /
```

Section 07 | Improvement

[데이터 불균형]

: 데이터 불균형이란 데이터 세트 내의 클래스의 분포가 불균형한 것을 의미한다. 데이터 불균형은 특정 클래스에 과적합 되는 현상을 유발할 수 있기 때문에 반드시 해결해야 하는 문제다.

• Weighted Random Sampling

```
# 각 클래스의 비율을 정하여 뽑힐 확률에 대한 가중치를 산정한다.
# 이를 기반으로 batch마다 class가 뽑힐 확률이 동일하도록 하는 과정!
# 모델의 학습 과정에서 균형 잡힌 데이터로 연산을 하도록

import torch
from torch.utils.data import DataLoader
import torchvision
import numpy as np

def make_weights_for_balanced_classes(img, nclasses):

    labels = []
    for i in range(len(img)):
        labels.append(img[i][1])

    label_array = np.array(labels)
    total = len(labels)

    count_list = []
    for cls in range(nclasses):
        count = len(np.where(label_array == cls)[0])
        count_list.append(total/count)

    weights = []
    for label in label_array:
        weights.append(count_list[label])

    return weights

# 데이터 세트 불러오기
trainset = torchvision.datasets.ImageFolder(root='./class', transform=transf)

weights = make_weights_for_balanced_classes(trainset.imgs, len(trainset.classes)) # 가중치 계산
weights = torch.DoubleTensor(weights) # 텐서 변환
sampler = torch.utils.data.sampler.WeightedRandomSampler(weights, len(weights)) # 샘플링 방법 정의

trainloader = DataLoader(trainset, batch_size=16, sampler=sampler) # 데이터 로더 정의
```

• Weighted Loss Function

```
# 데이터의 개수마다 LOSS의 크기를 다르게 설정해주는 것. -> 개수가 작으면 큰 weight를 주도록

import torch.nn as nn
import torch

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

```
num_ins = [40,45,30,62,70,153,395,46,75,194]
weights = [1-(x/sum(num_ins)) for x in num_ins] ##
class_weights = torch.FloatTensor(weights).to(device)

criterion = nn.CrossEntropyLoss(weight=class_weights)
```

• Data Augmentation

```
# weighted random sampling + augmentation 사용하면
# 사용하지 않는 것에 비해 효과가 일반적으로 좋음.

import torchvision.transforms as tr
import PIL

transf = tr.Compose([
    tr.ToPILImage(), tr.RandomCrop(60),
    tr.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
    tr.RandomHorizontalFlip(),
    tr.RandomRotation(10, resample=PIL.Image.BILINEAR),
    tr.ToTensor()
])
```

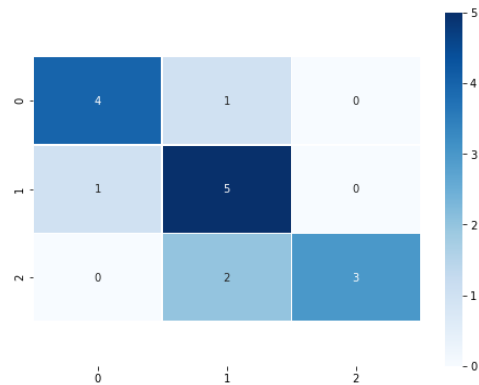
• Confusion Matrix

```
# 특정 class를 얼마나 잘 예측하는지
# -> 잘 못맞추는 class에 가중치 or Augmentation 시켜주기

from sklearn.metrics import confusion_matrix
import seaborn as sns
from matplotlib import pyplot as plt

actual = [1,1,1,1,1,1,0,0,0,0,2,2,2,2,2]
prediction = [1,1,1,0,1,1,0,0,1,0,2,2,2,1,1]

c_mat = confusion_matrix(actual, prediction) # 실제 라벨, 예측값
plt.figure(figsize = (8,6))
sns.heatmap(c_mat, annot=True, fmt="d", cmap='Blues', linewidths=.5)
b, t = plt.ylim()
b += 0.5
t -= 0.5
plt.ylim(b, t)
plt.savefig('confusion_matrix.png')
plt.show()
```



[Confusion Matrix 예시]

[과적합]

: 과적합은 학습 데이터에 치중하여 모델이 학습하는 현상으로 새로운 데이터에 대해서 대응을 못하는 문제다. 따라서 딥러닝에서 가장 쉽게 접할 수 있는 문제 유형이며 개선하기 힘든 문제다.

• Dropout & Batch Normalization

```
## 모델 내부에서 계산을 통해 과적합 방지
## Dropout : N개 중 xx%를 임의로 비활성화('0'으로 처리)
## B-N : convolution 연산이 끝난 후에 진행 /
      각 batch 마다의 평균과 표준편차를 계산해서 정규화

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.feature_extraction = nn.Sequential(nn.Conv2d(3, 6, 5),
                                                nn.BatchNorm2d(6),
                                                nn.ReLU(),
```

```

        nn.MaxPool2d(2, 2),
        nn.Conv2d(6, 16, 5),
        nn.BatchNorm2d(16),
        nn.ReLU(),
        nn.MaxPool2d(2, 2))

    self.classifier = nn.Sequential(nn.Linear(512, 120),
                                    nn.ReLU(),
                                    nn.Dropout(0.5), # 비활성화 시킬 노드의 비율
                                    nn.Linear(120, 64),
                                    nn.ReLU(),
                                    nn.Linear(64, 10))

    def forward(self, x):
        x = self.feature_extraction(x)
        x = x.view(-1, 512)
        x = self.classifier(x)

        return x

net = CNN().to(device) # 모델 선언

```

• L2 Regularization

```

# Loss function에 제약 ( optimizer에 설정 )

import torch.optim as optim

optimizer = optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-2)

```

• Data Augmentation

```

# Autoencoder, GAN, 사진 쪼개서 같이 합치는 모자이크.. etc!

# https://pytorch.org/docs/stable/torchvision/transforms.html
# 다양한 전처리 방법들을 확인할 수 있다.
import torchvision.transforms as tr
import PIL

# 기본적으로 torch 내에서 처리할 수 있는 방식들의 일부
transf = tr.Compose([
    [tr.ToPILImage(), tr.RandomCrop(60),
     tr.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
     tr.RandomHorizontalFlip(),
     tr.RandomRotation(10, resample=PIL.Image.BILINEAR),
     tr.ToTensor()]
])

```

• Label Smoothing

```

# 비슷한 성질을 가진 CLASS끼리의 간격을 줄여준다.
# LABEL을 ONE HOT VECTOR로 바꾼 후 값을 바꿔주는 작업이 PYTORCH에서 안 되어서 구현

import torch.nn as nn

class LabelSmoothingLoss(nn.Module):
    def __init__(self, classes, smoothing=0.0, dim=-1):
        super(LabelSmoothingLoss, self).__init__()
        self.confidence = 1.0 - smoothing
        self.smoothing = smoothing
        self.cls = classes
        self.dim = dim

    def forward(self, pred, target):
        pred = pred.log_softmax(dim=self.dim) # Cross Entropy 부분의 log softmax 미리 계산하기
        with torch.no_grad():
            # true_dist = pred.data.clone()
            true_dist = torch.zeros_like(pred) # 예측값과 동일한 크기의 영텐서 만들기
            true_dist.fill_(self.smoothing / (self.cls - 1)) # alpha/(K-1)을 만들어 줌 (alpha/K로 할 수도 있음)
            true_dist.scatter_(1, target.data.unsqueeze(1), self.confidence) # (1-alpha)y + alpha/(K-1)
        return torch.mean(torch.sum(-true_dist * pred, dim=self.dim)) # Cross Entropy Loss 계산

# model 안에서 loss를 정의할 때,
ls = LabelSmoothingLoss(10, smoothing=0.2)

```

[성능 개선]

• 과적합 | Early Stopping

- 라이브러리 및 데이터

```

import torch
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

```

```
import matplotlib.pyplot as plt

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

dataset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                       download=True, transform=transform)
trainset, valset = torch.utils.data.random_split(dataset, [30000, 20000])
trainloader = torch.utils.data.DataLoader(trainset, batch_size=32, shuffle=True)
valloader = torch.utils.data.DataLoader(valset, batch_size=32, shuffle=False)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=32, shuffle=False)
```

◦ 모델 정의

```
class ResidualBlock(nn.Module):

    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.stride = stride
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.conv_block = nn.Sequential(
            nn.Conv2d(self.in_channels, self.out_channels, kernel_size=3,
                      stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(self.out_channels),
            nn.ReLU(),
            nn.Conv2d(self.out_channels, self.out_channels, kernel_size=3,
                      stride=1, padding=1, bias=False),
            nn.BatchNorm2d(self.out_channels))

        if self.stride != 1 or self.in_channels != self.out_channels:
            self.downsample = nn.Sequential(
                nn.Conv2d(self.in_channels, self.out_channels,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.out_channels))

    def forward(self, x):
        out = self.conv_block(x)
        if self.stride != 1 or self.in_channels != self.out_channels:
            x = self.downsample(x)

        out = F.relu(x + out)
        return out

class ResNet(nn.Module):
    def __init__(self, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 64
        self.base = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU())
        self.layer1 = self._make_layer(64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(512, num_blocks[3], stride=2)
        self.gap = nn.AvgPool2d(4) # 4: 필터 사이즈
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, out_channels, num_blocks, stride):

        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            block = ResidualBlock(self.in_channels, out_channels, stride)
            layers.append(block)
            self.in_channels = out_channels

        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.base(x)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.gap(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out
```

◦ 모델 훈련

```
def modeltype(model):
    if model == 'resnet18':
        return ResNet([2, 2, 2, 2])

    elif model == 'resnet34':
        return ResNet([3, 4, 6, 3])

resnet = modeltype('resnet18').to(device)
#print(resnet)
PATH = './models/cifar_resnet_early.pth' # 모델 저장 경로
```

```

def validation_loss(data_loader):
    n = len(data_loader)
    running_loss = 0.0
    with torch.no_grad():
        resnet.eval() ## 학습 중간 과정에서 계산
        for data in data_loader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = resnet(images)
            loss = criterion(outputs, labels)
            running_loss += loss.item()
        resnet.train() # 계산이 끝난 후 TRAIN으로 변환
    return running_loss / n

train_loss_list = [] # 그래프를 그리기 위한 loss 저장용 리스트
val_loss_list = []
n = len(train_loader) # 배치 개수
early_stopping_loss = 1
for epoch in range(51):

    running_loss = 0.0
    for data in train_loader:

        inputs, labels = data[0].to(device), data[1].to(device) # 배치 데이터

        optimizer.zero_grad()
        outputs = resnet(inputs) # 예측값 산출
        loss = criterion(outputs, labels) # 손실함수 계산
        loss.backward() # 손실함수 기준으로 역전파 선언
        optimizer.step() # 가중치 최적화

        # print statistics
        running_loss += loss.item()

    train_loss = running_loss / n
    train_loss_list.append(train_loss)
    val_loss = validation_loss(val_loader)
    val_loss_list.append(val_loss)

    print('[%d] train loss: %.3f, validation loss: %.3f' %
          (epoch + 1, train_loss, val_loss))

    # LOSS가 가장 작은 MODEL이 SAVE 되도록!
    if val_loss < early_stopping_loss:
        torch.save(resnet.state_dict(), PATH)
        early_stopping_train_loss = train_loss
        early_stopping_val_loss = val_loss
        early_stopping_epoch = epoch

print('Final pretrained model >> [%d] train loss: %.3f, validation loss: %.3f' %
      (early_stopping_epoch + 1, early_stopping_train_loss, early_stopping_val_loss))

plt.plot(train_loss_list)
plt.plot(val_loss_list)
plt.legend(['train', 'validation'])
plt.title("Loss")
plt.xlabel("epoch")
plt.show()

```

◦ Test set에 적합

```

# 평가 데이터를 이용해 정확도를 구해보자.
# output은 미니배치의 결과가 산출되기 때문에 for문을 통해서 test 전체의 예측값을 구한다.

correct = 0
total = 0
with torch.no_grad():
    resnet.eval()
    for data in test_loader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = resnet(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0) # 개수 누적(총 개수)
        correct += (predicted == labels).sum().item() # 누적(맞으면 1, 틀리면 0으로 합산)

print('Test accuracy: %.2f %%' % (100 * correct / total))

# ResNet (overfitting): 75.57 %
# ResNet (early stopping): 76.22 %

```

Section 08 | Practice

```

# Main.py

import argparse # 터미널에서 실행할 수 있는 라이브러리
import training
import datasets
from tsne import tsne # tsne fn 그릴 수 있는 라이브러리

if __name__ == "__main__":

    parser = argparse.ArgumentParser(description='CIFAR10 image classification')
    # parser 선언
    parser.add_argument('--batch_size', default=128, type=int, help='batch size')

```

```

parser.add_argument('--epoch', default=101, type=int, help='training epoch')
parser.add_argument('--lr', default=1e-3, type=float, help='learning rate')
parser.add_argument('--l2', default=1e-4, type=float, help='weight decay')
# l2 => penalty
parser.add_argument('--model_name', default='resnet18', type=str, help='model name')
# 모델이 여러 개일 경우 고를 수 있음
parser.add_argument('--pretrained', default=None, type=str, help='model path')
# pretrained model 일 경우 경로 지정 ("default 변수")
parser.add_argument('--train', default='train', type=str, help='train and eval')
args = parser.parse_args()
print(args)

# 데이터 불러오기 => train과 test 따로 가져옴
trainloader, testloader = datasets.dataLoader(args.batch_size)
print('Completed loading your datasets.')

# 모델 불러오기 및 학습하기 => 학습하면서 모델을 생성할 수 있도록 함
learning = training.SupervisedLearning(trainloader, testloader, args.model_name, args.pretrained)

if args.train == 'train':
    learning.train(args.epoch, args.lr, args.l2)
else:
    train_acc = learning.evaluation(trainloader)
    test_acc = learning.evaluation(testloader)
    print(f' Train Accuracy: {train_acc}, Test Accuracy: {test_acc}')

# t-SNE graph
tsne(testloader, args.model_name, args.pretrained)

```

```

# Datasets.py

import torchvision
import torchvision.transforms as tr
from torch.utils.data import DataLoader

# Data loader
def dataLoader(batch_size):

    # train에 대한 전처리 정의

    transf = tr.Compose([tr.RandomCrop(32, padding=4), tr.RandomHorizontalFlip(), tr.ToTensor(),
                        tr.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))])

    # (32 * 32) 픽셀 이미지 random crop
    # Normalize => 정규화 ~ cifar10 평균과 표준편차 : 항상 학습속도가 향상되는 것은 아님

    test_transf = tr.Compose([tr.ToTensor(), tr.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))])
    # test => data augmentation 진행 안함

    trainset = torchvision.datasets.CIFAR10(root='./data', download=True, train=True, transform=transf)
    trainloader = DataLoader(trainset, batch_size=batch_size, num_workers=0)

    testset = torchvision.datasets.CIFAR10(root='./data', download=True, train=False, transform=test_transf)
    testloader = DataLoader(testset, batch_size=batch_size, shuffle=False, num_workers=0)

    # loader => 배치까지 형성에 외부로 내보내는 기능
    # 데이터가 여러 개인 경우 조건문(if)을 사용해 선택할 수 있게 구성할 수 있음
    return trainloader, testloader

```

```

# Models.py

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

# Residual Block 정의
class ResidualBlock(nn.Module):

    def __init__(self, in_planes, planes, stride=1):
        super(ResidualBlock, self).__init__()

        self.conv_block = nn.Sequential(
            nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(planes),
            nn.ReLU(),
            # BatchNormalization, Relu => 성능 향상
            nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(planes)
        )

        # stride가 1인 경우, usually padding 값도 1
        # output size = 1 + (input size + 2 * padding - filter size) / stride
        # ouput size == input size

        self.downsample = nn.Sequential()
        if stride != 1 or in_planes != planes:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_planes, planes, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes)
            )

        # downsample
        # 이미지 병목현상(사이즈가 줄어드는 현상) 이후 x와 output의 사이즈를 동등하게 맞춤
        # .. stride = 2인 경우 피쳐맵의 사이즈(out)는 입력 사이즈(x)보다 작아짐
        # 따라서 이 경우에는 x의 사이즈를 out 사이즈에 맞춰야만 함
        # 입력 채널 수(in_planes)가 출력 채널 수(planes)보다 적은 경우에도 해당됨
        # downsample을 통해 x와 out 채널 수를 맞춤

    def forward(self, x):
        out = self.conv_block(x)
        out += self.downsample(x)

```

```

# Residual Block의 skip connection
# skip connection => 이전 정보를 건너뛰어 반영하는 행위

out = F.relu(out)
return out

class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_planes = 64

        self.base = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU()
        )
        # Residual Block 이전의 Convolutional Layer 정의
        # Maxpooling을 사용하지 않는 이유 => 이미지 자체가 너무 작아지면 안되기 때문

        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)
        # 각각의 layer에는 block이 2개씩 들어있음

        self.gap = nn.AvgPool2d(4)
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            # 첫 번째 strides = [1 1] = [1] + [1] * 1
            # 두 번째 strides = [2 1] = [2] + [1] * 1
            layers.append(block(self.in_planes, planes, stride))
            # block(64, 64, 1), block(64, 64, 1)
            # block(64, 128, 2), block(128, 128, 1)
            # block(128, 256, 2), block(256, 256, 1)
            # block(256, 512, 2), block(512, 512, 1)
            self.in_planes = planes
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.base(x)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.gap(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

    def modeltype(model):
        if model == 'resnet18':
            return ResNet(ResidualBlock, [2, 2, 2, 2])

        elif model == 'resnet34':
            return ResNet(ResidualBlock, [3, 4, 6, 3])

```

[ResNet]

- 구성하는 기본 요소 : residual block \Rightarrow 연속적인 형태
 - convolutional layer block 중 이전의 값을 찾기 위한 모델
- ▼ img

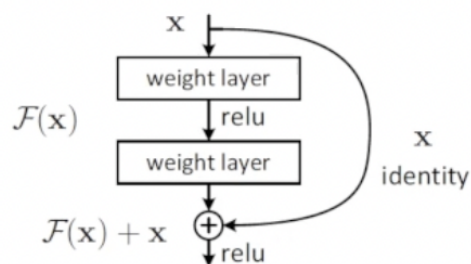


Figure 2. Residual learning: a building block.

- 18-layer ~ 152-layer (layer 수)
- 장점 \Rightarrow Back-propagation 시 vanishing gradient 방지 ~ 다른 모델에서도 사용

```

# Training.py
import torch
import torch.nn as nn

```

```

import torch.optim as optim
import models
from matplotlib import pyplot as plt
from tqdm import tqdm # for loop을 돌 때 진행상황을 알려줌 (시각적)

class SupervisedLearning():

    def __init__(self, trainloader, testloader, model_name, pretrained):
        # testset과 model은 이미 정의됨

        self.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        self.trainloader = trainloader
        self.testloader = testloader

        self.model_name = model_name
        self.model = models.modeltype(self.model_name)
        self.model = self.model.to(self.device)

        if pretrained != None:
            self.model.load_state_dict(torch.load(pretrained))
            print('Completed you pretrained model.')

        print('Completed loading your network.')

        self.criterion = nn.CrossEntropyLoss()

    def evaluation(self, dataloader):

        correct = 0
        total = 0
        self.model.eval()

        with torch.no_grad():
            for data in dataloader:
                images, labels = data[0].to(self.device), data[1].to(self.device)
                outputs = self.model(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        acc = 100 * correct / total

        return acc

    def train(self, epoch, lr, l2):

        optimizer = optim.Adam(self.model.parameters(), lr=lr, weight_decay=l2)

        train_loss_list = []
        test_loss_list = []
        n = len(self.trainloader)
        m = len(self.testloader)
        test_loss = 10 # dummy

        print("Start training the model.")
        for epoch in tqdm(range(epoch)):

            running_loss = 0.0

            for data in self.trainloader:

                inputs, labels = data[0].to(self.device), data[1].to(self.device)
                optimizer.zero_grad()
                outputs = self.model(inputs)
                trainloss = self.criterion(outputs, labels)

                trainloss.backward()
                optimizer.step()

                running_loss += trainloss.item()

            train_cost = running_loss / n
            train_loss_list.append(train_cost)

            running_loss = 0.0

            with torch.no_grad():
                for data in self.testloader:

                    inputs, labels = data[0].to(self.device), data[1].to(self.device)
                    outputs = self.model(inputs)
                    testloss = self.criterion(outputs, labels)
                    running_loss += testloss.item()

                test_cost = running_loss / m
                test_loss_list.append(test_cost)

            if epoch % 10 == 0:
                print(f'Epoch {epoch}, Train Loss: {train_cost}, Test Loss: {test_cost}')

            if test_cost <= test_loss:
                torch.save(self.model.state_dict(), './results/' + self.model_name + '_best.pth')
                test_loss = test_cost
                best_epoch = epoch

        torch.save(self.model.state_dict(), './results/' + self.model_name + '_last.pth')
        print('Finished Training')

        # Graph
        plt.figure(figsize=(8, 5))
        plt.plot(train_loss_list)
        plt.plot(test_loss_list)
        plt.legend(['Train Loss', 'Test Loss'])
        plt.savefig('./results/' + self.model_name + '_graph.png')

        self.model.load_state_dict(torch.load('./results/' + self.model_name + '_best.pth'))
        train_acc = self.evaluation(self.trainloader)

```



```
test_acc = self.evaluation(self.testloader)
print(f'Epoch{best_epoch}: Train Accuracy: {train_acc}, Test Accuracy: {test_acc}')
```

```
# Tsne.py

# fully connected 이전 값으로 tsne 그림

from sklearn.manifold import TSNE
import numpy as np
from matplotlib import pyplot as plt
import models
import torch
import torch.nn as nn

class Identity(nn.Module):
    def __init__(self):
        super(Identity, self).__init__()

    def forward(self, x):
        return x

def tsne(dataloader, model_name, pretrained):

    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model_name = model_name
    model = models.modeltype(model_name)
    model = model.to(device)
    model.load_state_dict(torch.load(pretrained))
    model.fc = Identity() # ** Identity fn 바꿔주기

    actual = []
    deep_features = []

    model.eval()
    with torch.no_grad():
        for data in dataloader:
            images, labels = data[0].to(device), data[1].to(device)
            features = model(images)

            deep_features += features.cpu().numpy().tolist()
            actual += labels.cpu().numpy().tolist()

    tsne = TSNE(n_components=2, random_state=0)
    cluster = np.array(tsne.fit_transform(np.array(deep_features)))
    actual = np.array(actual)

    plt.figure(figsize=(10, 10))
    cifar = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
    for i, label in zip(range(10), cifar):
        idx = np.where(actual == i)
        plt.scatter(cluster[idx, 0], cluster[idx, 1], marker='.', label=label)

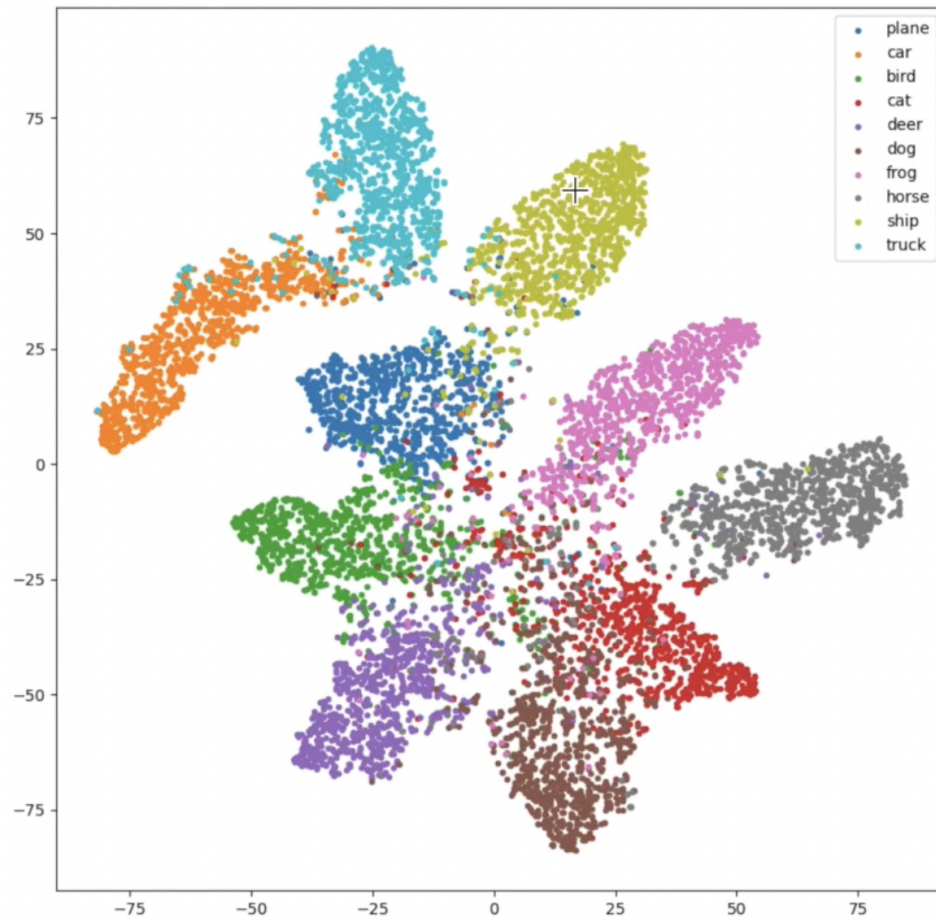
    plt.legend()
    plt.savefig('./results/'+ model_name+'_tsne.png')
```

[터미널에서 실행하기 및 결과 분석]

- GPU로 돌리기 위해 Google Colab의 “런타임 유형변경”에서 GPU 선택
 - 현재 경로 입력
 - 앞에 ! 붙이기 (ex !python ~.py / !sh ~.sh)
 - 평가할 경우 train=“eval”로 변경 ~ “--”로 변수 수정 (ex --train eval)
- * 적절한 augmentation과 model 사용 시 성능 향상

[수행한 내용]

- resnet18 best
 - resnet18 last
 - tsne graph
 - 분류가 되지 않은 데이터 역추적 → 오분류 확인
 - 캠 이미지 → 오분류 확인
 - data augmentation 강화
- ▼ img



- loss graph \Rightarrow test와 train 둘 다 일정 수준으로 수렴
 - overfitting 발생 ; train loss 값만큼 test loss 값을 내릴 수 있게 튜닝 필요
 - 1) Label smoothing을 통해 경향성 확인
 - 2) l2 regularization의 penalty 높이기
 - 3) data augmentation 적용
 - 4) 새로운 model로 변경 \Rightarrow 여러 개의 모델로 검증
 - * cross validation \Rightarrow 더욱 엄격한 평가
 - \rightarrow train / validation / test 세 단계로 진행

▼ img

