3️⃣

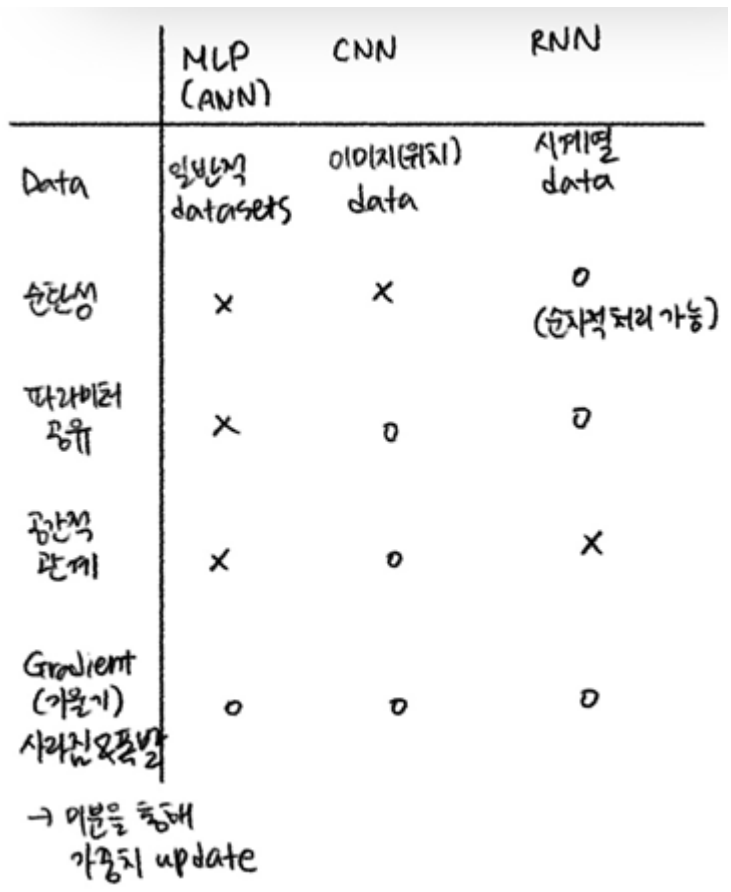
딥러닝_실습(CNN)

1. 인공신경망이란?

- 인공신경망이란
  - 인간의 뇌에 있는 뉴런에서 영감을 받아 고안한 모델
- 머신러닝과 딥러닝
  - 머신러닝과 딥러닝을 구분하는 이유
    - 딥러닝>자연어처리, 컴퓨터비전>Object Retection → 모델이 너무 많음
    - 머신러닝

      input→feature extraction(특징추출,인간이 직접 만든 feature)→classification→output
    - 딥러닝

      input→feature extraction+Classification→output (특징추출 생략, 아예 없진 않음)
    - 신경망 모델(딥러닝)은 특징 추출 과정 모델 내부에서 전부 처리하려고함
      - 특징 추출 편한 데이터는 머신러닝 방법
- 종류
  - ANN(Artificial Neural Network) : 모든 데이터가 한번에 다음 층에 연결되는 All-to-all 관계를 강조하는 신경망 (but, 지역적 정보 잃어버림) = MLP
  - CNN(Convolution Neural Network) : 데이터의 지역적인 특징을 고려하는 Locality 관계를 강조하는 신경망 → filter 단위
  - RNN(Recurrent Neural Network) : 데이터의 순차적인 특징을 고려하는 Sequentiality(순차성) 관계 → 은닉층의 정보 다음 output에도 전파



실습_pandas, numpy, pytorch

```
import numpy as np
import pandas as pd

import torch
```

```
import torch.nn as nn

### pandas, numpy, pytorch의 차이

from sklearn.datasets import load_iris #colab 환경에서는 데이터가 없기 때문에 sklearn을 통해 데이터를 가져

iris=load_iris()

iris

iris.keys() #iris의 ket값 확인

### numpy의 array 자료형 : 모델이 받아들이기 편하게 데이터를 구성해서 보여주는 자료형

iris_data=iris['data']
print(type(iris_data))
iris_data

iris_colname=iris['feature_names']  #칼럼명
iris_colname

### pandas의 dataframe 구조 : 열이름을 통해 해당 숫자들이 어떤 의미를 갖는지 보여줌

iris_pandas=pd.DataFrame(iris_data,columns=iris_colname) #입력변수
print(type(iris_pandas))
iris_pandas

## pytorch : tensor라는 자료구조를 통해서 데이터를 gpu에 올려서 딥러닝 처리가 가능하게 해줍니다.
## numpy랑 동일하게 둘다 모델이 처리하게 편하게 데이터를 보여준다.
## 서로 변환이 쉽다. 사용되는 문법도 굉장히 비슷하다.

iris_pytorch=torch.tensor(iris_data) #pandas -> tensor
print(type(iris_pytorch))
iris_pytorch

torch.tensor(iris_pandas) ##tensor에 pandas dataframe과 연동이 안됨 -> error

# numpy와 tensor는 서로 변환되기 때문에 쉽게 변환해주는 메소드(함수)를 가지고 있음

iris_pytorch.numpy() #tensor->numpy

torch.from_numpy(iris_data) #tensor->numpy
```

- 인공신경망을 colab에서 구현할 때 pytorch library에 tensor형태로 구현
- pandas와 tensor는 연동 안됨

2. 인공 신경망을 학습하는 방법

- 순전파(Feed Forward) : 가중치(weight) 부여 → 가중합 계산(bias도 합함) → 활성화 함수 적용
  - 활성화 함수 : 모델의 비선형성을 더해 주는 역할, 모델의 복잡도가 높아짐, 적절한 복잡도를 통해 데이터를 잘 표현하는 모델 선택하기 위해
  - 비선형성 : 선형으로 분리할 수 없는 영역을 선형으로 분리할 수 있도록 함

실습_순전파

```
### 순전파 실습 : 가중치 부여와 가중합 계산
```

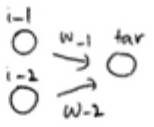
```
torch.manual_seed(1017)  # random한 요소들을 제어하는 장치

network = nn.Linear(in_features=3, out_features=7)
#입력으로 길이가 3인 tensor를 받아서, 출력으로 길이가 7인 tensor를 출력하는 네트워크

torch.manual_seed(1017) #고정을 하려면 seed를 shell마다 넣어줘야됨

inputs=torch.randn(3) #랜덤한 숫자를 생성하는 메소드 -> 길이 3짜리인 tensor 만들기 위함
inputs

network(inputs)
# 위 값이 어떻게 출력된걸까?

network.weight #두번째층 하나의 노드로 가는 3개의 가중치

torch.sum(inputs*network.weight[0]) #하나의 노드에 대한 weight -> 0.0314가 나와야함 -> 네트워크값이랑 다르

network #bias가 더해졌기 때문 값이 다르게 출력

torch.manual_seed(1017)

network_nobias=nn.Linear(in_features=3, out_features=7,bias=False)
network_nobias.weight

network_nobias(inputs)

#그럼 모든 노드들이 같은 bias값을 가질까?

bias_first_output=network(inputs)[0] #bias가 있는 네트워크의 첫번째 출력값
nobias_first_output=network_nobias(inputs)[0] #bias가 없는 네트워크의 첫번째 출력값
bias_first_output-nobias_first_output #첫번째 node에 대한 bias값

bias_second_output=network(inputs)[1] #bias가 있는 네트워크의 두번째 출력값
nobias_second_output=network_nobias(inputs)[1] #bias가 없는 네트워크의 두번째 출력값
bias_second_output-nobias_second_output #두번째 node에 대한 bias값

import numpy as np
import pandas as pd

import torch
import torch.nn as nn

### 순전파 실습 : 활성화 함수의 적용

network #앞서 만들어놓은거 그대로 가져온 네트워크

def sigmoid(x):
  return 1/(1+ np.exp(-x)) #수식(numpy)

outputs=network(inputs)
sigmoid(outputs) #array가 아닌 tensor -> sigmoid는 numpy로 만든 함수기 때문에 tensor 못 받음

def sigmoid_torch(x):
  return 1/(1+ torch.exp(-x)) #수식(tensor)

sigmoid_torch(outputs) #grad_fn : gradient function (가중치 update할 때에는 역전파로 오른쪽에서 왼쪽으로

outputs_nograd=outputs.detach() #역전파 과정에 필요한 기울기를 제거하는 함수
```

```
sigmoid(outputs_nograd.numpy()) #torch를 numpy로 변환을 하면 기존에 만든 함수가 잘 작동 (sigmoid함수는 nu

np.round(sigmoid(outputs_nograd.numpy()),4)

torch.manual_seed(1017)

total_network=nn.Sequential(
    nn.Linear(3,7),
    nn.Sigmoid()
) # sequential안에 내가 사용하고 싶은 신경망, 활성화함수 등 넣으면 모두 출력가능 / sigmoid함수 네트워크형태로 사

total_network(inputs)

### 활성화 함수의 효과 확인하기 : 비선형성 증가

from sklearn.datasets import load_iris

iris=load_iris()
iris_data=iris['data']
iris_label=iris['target'] #label(붓꽃의 종류)
iris_colname=iris['feature_names'] #dataframe을 만들기 위해서 가져온 열 이름

print(iris_data.shape,iris_label.shape) #iris_label 1차원 배열

iris_labeldata=np.hstack([iris_data, iris_label.reshape(-1,1)]) #iris_label reshape(a, b) = 데이
iris_df=pd.DataFrame(iris_labeldata, columns=iris_colname+['label'])
iris_df

iris_df.iloc[:,[0,2]] # 인덱싱

iris_length=iris_df.iloc[:,[0,2,4]] # 이해하기 쉬운 그림을 그리기 위해 변수를 2개로 축소

import matplotlib.pyplot as plt

plt.scatter(iris_length['sepal length (cm)'],iris_length['petal length (cm)'], c=iris_length['la

sig_iris=sigmoid(iris_length.iloc[:,[0,1]])
plt.scatter(sig_iris['sepal length (cm)'], sig_iris['petal length (cm)'], c=iris_length['label']
```

- bias가 가중치 계산될 때 합쳐져서 계산되어 출력값으로 출력
- 활성화함수(sigmoid) : numpy로 만든 함수쓰려면 gradient function으로 가중치 update할 때 역전파 과정을 거치므로 이때 numpy가 연동이 안되므로 detach()를 통해 역전파 과정에 필요한 기울기 제거 후 함수 적용하면 됨

- 역전파(Backpropagation) : 정답과 다른 정도를 기준 삼아 가중치를 업데이트(모델이 정답을 향해 찾아가는 과정)

<역전파>



i_1=0.3 i_2=0.7

w_1=0.6 w_2=0.9

tar=1.5

(원하는 output)

pred= i_1 * w_1 + i_2 * w_2 (가정] 활성화x, 편향x)

= 0.3*0.6+0.7*0.9

= 0.18+0.63 = 0.81

tar-pred = 1.5-0.81= 0.69 =loss (정답과 다를 정도)

loss= tar-pred = tar-(i_1*w_1+i_2*w_2)

tar-i_1*w_1-i_2*w_2

↓

w_1에 대한 편미분 : $-i_1 = -0.3$

w_2에 대한 : $-i_2 = -0.7$

Update되는 공식 : optimizer = SGD optim

(update 된) (예전가중치) (학습률)

w_up = w_past - lr * 편미분

[가중치1번 update] - lr=0.01

w1_up = 0.6-(0.01*(-0.3))=0.6-(-0.003)=0.603

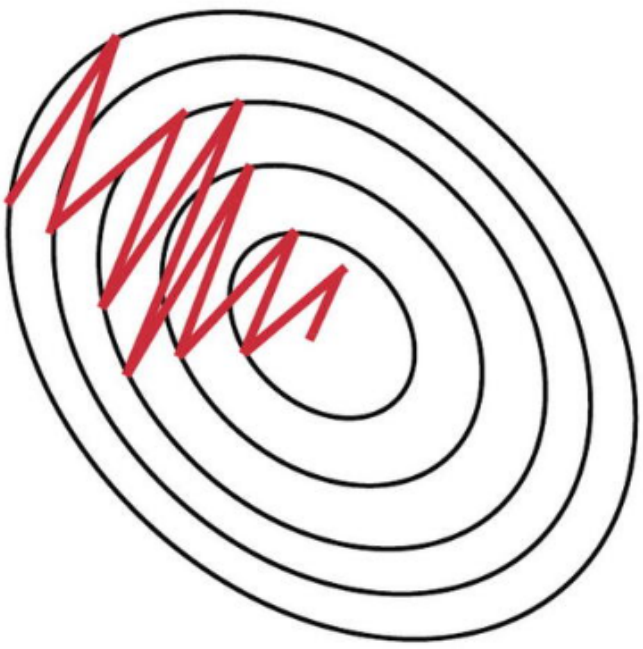
w2_up = 0.9-(0.01*(-0.7))=0.9-(-0.007)=0.907

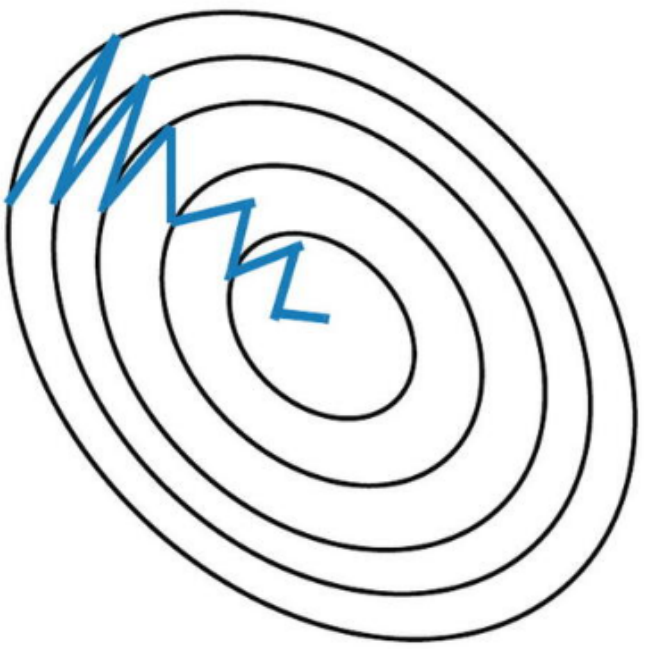
pred_up = 0.3*0.603+0.7*0.907=0.8158

loss_up= 1.5-0.8158= 0.6842

loss=0.69 > loss_up=0.6842

- loss function : 오답을 정의
- optimizer : 더 깊은 곳 (정답을 향해) 찾아감
- 학습률(learning_rate)





→ momentum은 잘 안돌아다니고 중앙을 향해 갈 수 있게 해줌

→ 현재 adam이라는 optimizer 가장 많이 쓰임 (성능 가장 좋음)

실습_역전파

```
### 역전파 : 가중치 업데이트

torch.manual_seed(1017)
```

```
inputs=torch.randn(2,2) #2행2열짜리 생성
inputs

torch.manual_seed(777)

targets=torch.rand(2) #label 생성
targets

torch.manual_seed(1017)

total_network=nn.Sequential(
    nn.Linear(2,1),
    nn.Sigmoid()
)

total_network[0].weight #input의 weight

learning_rate=0.001
criterion=nn.MSELoss()
optimizer=torch.optim.Adam(total_network.parameters(), lr=learning_rate) #학습률, loss함수, optimi

total_network.train()

pred=total_network(inputs)
print(pred)

loss=criterion(targets, pred)
print(loss)

loss.backward()  #loss값을 참고해서 각 가중치에 대해서 편미분을 하겠다.
optimizer.step() #편미분된 값과 미리 지정한 학습률을 통해 한 스텝 최적화 진행

total_network[0].weight

pred=total_network(inputs)
print(pred)

loss=criterion(targets,pred)
print(loss)
```

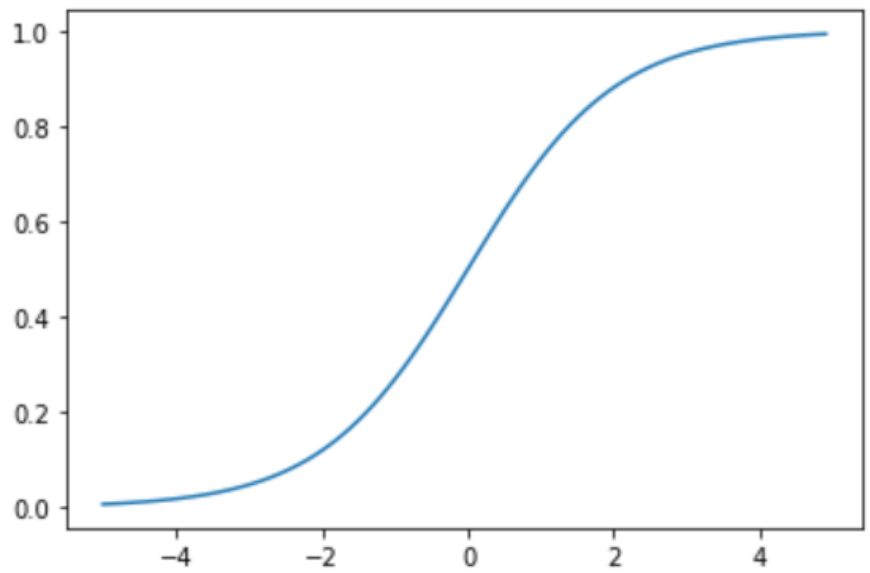
3. 인공 신경망의 문제점

1. 기울기 소실/폭발 : Vanishing/Exploding Gradient
    - 가중치가 업데이트되지 않아 학습이 불가능한 문제 → sigmoid활성화함수 조금만 0에 치중돼도 가중치 update가 안돼 학습이 안됨
2. 과적합(Overfitting) : 머신러닝의 고질적인 문제점
    - 학습데이터에 과적합되어 테스트 데이터에서는 성능이 낮게 나오는 현상
    - 모델이 복잡해질수록 overfitting될 확률 높아짐
3. Black Box형 구조 : 인공 신경망은 해석이 불가능 : 설명력이 부족(숫자들이 정확히 의미하는것 모름) → 그럼 다른 머신러닝은 해석 가능?

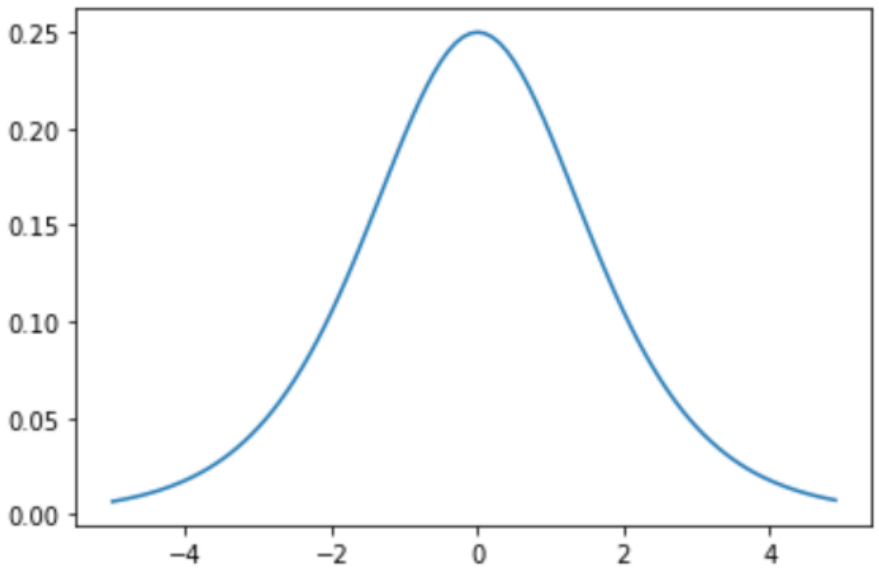
실습_인공신경망 문제점

```
### 기울기가 소실/폭발 문제 확인

import numpy as np
import pandas as pd

import torch
import torch.nn as nn
import matplotlib.pyplot as plt

def sigmoid(x):
  return 1/(1+np.exp(-x))

x=np.arange(-5,5,0.1)
y=sigmoid(x)
plt.plot(x,y) #sigmoid함수의 개형 (-1,1)로 범위 한정됨

diff_y=sigmoid(x)*(1-sigmoid(x))
plt.plot(x,diff_y) #값 (0,0.25)사이 -> 기울기 소실/폭발
```

```
[<matplotlib.lines.Line2D at 0x7f51a7f9ae50>]
```




```
[<matplotlib.lines.Line2D at 0x7f51a7afd950>]
```



4. 인공 신경망의 문제점 극복 방법

1-1. 기울기 소실/폭발 : 활성화 함수

- 활성화 함수 부분을 수정하여 해결 → 미분했을 때 적당한 기울기를 가지는 함수를 적용
  - sigmoid → ReLU

1-2. 기울기 소실/폭발 : learning rate(1step 간격) & initialize weight(가중치 초기화:처음 가중치 알맞게)

- 학습률 조정하여 해결 → 학습률을 통해 가중치가 업데이트 되는 간격을 조절 가능

1-3. 기울기 소실/폭발 : Batch Normalization

- 데이터분포 조절하여 학습이 잘되도록 만듦
- batch : 큰 데이터셋을 batch로 나누어 모델에 들어갈 수 적용하는 것
- 활성화함수 전에 batch normalization 적용

2-1. overfitting : weight decay & drop-out

- L2 regularization : loss를 늘림 (penalty)
- 노드를 일정 비율 사용하지 않는 방법 → 모델이 보다 간결해짐

2-2. overfitting : cross validation(교차검증)

- 모든 데이터가 빠짐없이 학습 과정과 검증 과정에 참여할 수 있도록 데이터를 분할하는 방법
- 학습데이터가 계속 바뀌므로 과적합될 확률 줄여줌

3. Black Box형 구조 : XAI
    - 기울기의 정도를 시각화하여 설명하는 방법

실습_기울기 소실/폭발

```
### 기울기 소실/폭발 : 활성화 함수를 변경하여 해결

torch.manual_seed(1017)

inputs=torch.randn(2,2) #2행 2열짜리 데이터

sigmoid_network=nn.Sequential(
    nn.Linear(2,1),
    nn.Sigmoid()
) #model

torch.manual_seed(777)

targets=torch.randn(2)

learning_rate=0.001 #학습률
criterion=nn.MSELoss()
optimizer=torch.optim.Adam(sigmoid_network.parameters(), lr=learning_rate)

for epoch in range(0,100): # 한번 학습하는걸 epoch, 총 100번 학습
  print('***Start {} Epoch'.format(epoch+1)) #epoch마다 시작되는 걸 보겠다 -> 중괄호안에 들어감
  sigmoid_network.train()

  pred=sigmoid_network(inputs) # 예측값 할당

  loss=criterion(targets, pred) # loss값 계산
  print('Loss : {}'.format(loss)) # loss값을 에포크마다 보겠다

  optimizer.zero_grad() # 기울기가 누적이 되는걸 방지
  loss.backward() #loss값을 통해서 편미분
  optimizer.step() # 가중치를 한스텝 업데이트

  print('Weight : {}'.format(sigmoid_network[0].weight),'\n') # 가중치를 epoch마다 보겠다

torch.manual_seed(1017)

inputs=torch.randn(2,2) #2행 2열짜리 데이터

relu_network=nn.Sequential(
    nn.Linear(2,1),
    nn.ReLU()  #relu함수
)

learning_rate=0.001 #학습률
criterion=nn.MSELoss()
optimizer=torch.optim.Adam(relu_network.parameters(), lr=learning_rate)

for epoch in range(0,100): # 한번 학습하는걸 epoch, 총 100번 학습
```

```
  print('***Start {} Epoch'.format(epoch+1)) #epoch마다 시작되는 걸 보겠다 -> 중괄호안에 들어감
  relu_network.train()

  pred=relu_network(inputs) # 예측값 할당

  loss=criterion(targets, pred) # loss값 계산
  print('Loss : {}'.format(loss)) # loss값을 에포크마다 보겠다

  optimizer.zero_grad() # 기울기가 누적이 되는걸 방지
  loss.backward() #loss값을 통해서 편미분
  optimizer.step() # 가중치를 한스텝 업데이트

  print('Weight : {}'.format(relu_network[0].weight),'\n') # 가중치를 epoch마다 보겠다

  ####Sigmoid는 기울기 소실로 인해 가중치 update가 많이 되지 않음 -> relu함수로 인해 개선####

### 기울기 소실/폭발 : 학습률을 변경하여 해결

torch.manual_seed(1017)

inputs=torch.randn(2,2) #2행 2열짜리 데이터

sigmoid_network=nn.Sequential(
    nn.Linear(2,1),
    nn.Sigmoid()
)

learning_rate=0.01 #학습률 0.001 -> 0.1 (step이 더 크게)
criterion=nn.MSELoss()
optimizer=torch.optim.Adam(sigmoid_network.parameters(), lr=learning_rate)

for epoch in range(0,100): # 한번 학습하는걸 epoch, 총 100번 학습
  print('***Start {} Epoch'.format(epoch+1)) #epoch마다 시작되는 걸 보겠다 -> 중괄호안에 들어감
  sigmoid_network.train()

  pred=sigmoid_network(inputs) # 예측값 할당

  loss=criterion(targets, pred) # loss값 계산
  print('Loss : {}'.format(loss)) # loss값을 에포크마다 보겠다

  optimizer.zero_grad() # 기울기가 누적이 되는걸 방지
  loss.backward() #loss값을 통해서 편미분
  optimizer.step() # 가중치를 한스텝 업데이트

  print('Weight : {}'.format(sigmoid_network[0].weight),'\n') # 가중치를 epoch마다 보겠다
```

실습_과적합 방지

```
### MNIST 실습 및 과적합 확인하기

import numpy as np
import pandas

import matplotlib.pyplot as plt
import matplotlib.cm as cm

import torch
```

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

from torch.utils.data import DataLoader

from sklearn.datasets import fetch_openml

mnist=fetch_openml('mnist_784') #28x28 size의 이미지를 늘려놓음
mnist_data=mnist.data

mnist_data

mnist_ex=np.array(mnist_data.iloc[1]).reshape(28,28)
plt.imshow(mnist_ex, cmap=cm.gray) #cmap은 색깔

# pytorch로 넘어와서 다시 다운로드
from torchvision import datasets #데이터셋
from torchvision import transforms #데이터셋 변형

download_root='./MNIST'
train_dataset=datasets.MNIST(root=download_root,
                             train=True,
                             transform=transforms.ToTensor(),
                             download=True)

valid_dataset=datasets.MNIST(root=download_root,
                             train=False,
                             transform=transforms.ToTensor(),
                             download=True)

batch_size=512

train_loader=DataLoader(train_dataset,
                        batch_size=batch_size,
                        #shuffle=True -> dataset 섞어줌
                        )
valid_loader=DataLoader(valid_dataset,
                        batch_size=batch_size) #valid는 섞으면 안됨

torch.manual_seed(1017)

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

    self.network=nn.Sequential( #모델 선언
        nn.Linear(28*28, 256), #노드 1->2
        nn.ReLU(),
        nn.Linear(256,128), #노드 2->3
        nn.ReLU(),
        nn.Linear(128,10), #노드 3->4, 출력층 node 10개
        nn.Softmax() #분류문제 -> 출력층 노드의 합을 1로 만들어줌(확률)
    )

  def forward(self, x):#순전파
    pred=self.network(x)
    return pred
```

```
model=model()
learning_rate=0.001
optimizer=torch.optim.Adam(model.parameters(), lr=learning_rate)

criterion=nn.CrossEntropyLoss()
num_epochs=2000

torch.manual_seed(1017)

for epoch in range(num_epochs): #epoch 설정
  model.train() #학습 모드
  total_train_loss=0  #사용할 변수를 미리 할당
  total_train_acc=0   #사용할 변수를 미리 할당
  for img, labels in train_loader:  #한번 돌아가면(이미지를 꺼내면) 512장씩 꺼내겠다
    img=img.reshape(-1,784) #데이터를 (장수, 길이)의 shape으로 변환

    pred=model(img)
    loss=criterion(pred,labels)

  total_train_loss += loss.item() #epoch 하나당 loss를 계산하기 위해서 batch별로 loss를 누적

  optimizer.zero_grad()
  loss.backward()
  optimizer.step()

  total_train_acc += (torch.argmax(pred, dim=1)==labels).sum() / img.size(0) #argmax()->예측값C

model.eval() #평가 모드 : 역전파가 이루어지지 않아요
total_valid_loss=0
total_valid_acc=0

with torch.no_grad(): # 미분을 진행하지 않겠다 (누적되는 기울기 자체가 없다)
  for img, labels in valid_loader:
    img=img.reshape(-1,784)
    pred=model(img)

    loss=criterion(pred, labels)
    total_valid_loss+=loss.item()
    total_valid_acc += (torch.argmax(pred, dim=1)==labels).sum() / img.size(0)

    train_loss=total_train_loss / len(train_loader) #loader의 길이 = 전체 데이터수 / batch_size
    valid_loss=total_valid_loss / len(valid_loader)
    train_acc=total_train_acc / len(train_loader)
    valid_acc=total_valid_acc / len(train_loader)

    print('***Epoch: {}/{}, Train Loss & Accuracy: ({:.3f},{:.3f}),Valid Loss & Accuracy: ({:.3f

### 과적합 방지 : Drop-out, Epoch 고정되었을 때

torch.manual_seed(1017)

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

    self.network==nn.Sequential( #모델 선언
        nn.Linear(28*28, 256), #노드 1->2
        nn.ReLU(),
```

```
        nn.Dropout(0,2),    #활성화함수 뒤쪽 -> 20%만큼의 노드를 사용하지 않겠다
        nn.Linear(256,128), #노드 2->3
        nn.ReLU(),
        nn.Dropout(0,2),
        nn.Linear(128,10), #노드 3->4, 출력층 node 10개
        nn.Softmax()) #분류문제 -> 출력층 노드의 합을 1로 만들어줌(확률)

  def forward(self, x):#순전파
    pred=self.network(x)
    return pred

model = model()
learning_rate = 0.001
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

torch.manual_seed(1017)

for epoch in range(num_epochs):
    model.train()
    total_train_loss = 0
    total_train_acc = 0
    for img, labels in train_loader: # dataloader는 image와 label로 구성
        img = img.reshape(-1, 784)

        pred =  model(img)
                #print(torch.argmax(pred, dim=1))
        loss = criterion(pred, labels)

        total_train_loss += loss.item()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_train_acc += (torch.argmax(pred, dim=1) == labels).sum() / img.size(0)

    model.eval()
    total_valid_loss = 0
    total_valid_acc = 0

    with torch.no_grad(): # 미분을 진행하지 않는다는 의미
        for img, labels in valid_loader:
            img = img.reshape(-1, 784)
            pred =  model(img)

            loss = criterion(pred, labels)

            total_valid_loss += loss.item()
            total_valid_acc += (torch.argmax(pred, dim=1) == labels).sum() / img.size(0)

    train_loss = total_train_loss / len(train_loader)
    valid_loss = total_valid_loss / len(val_loader)
    train_acc = total_train_acc / len(train_loader)
    valid_acc = total_valid_acc / len(val_loader)

    print("***Epoch : {}/{}, Train Loss & Accuracy: ({:.3f}, {:.3f}), Valid Loss & Accuracy: ({

### 얼리스탑을 통해 과적합 방지하기 : epoch를 고정시키지 않았을 때
```

```
- 얼리스탑은 과적합을 판단하면 학습을 중지
- epoch를 얼만큼 주든 상관이 없음

torch.manual_seed(1017)

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

    self.network==nn.Sequential( #모델 선언
        nn.Linear(28*28, 256), #노드 1->2
        nn.ReLU(),
        nn.Linear(256,128), #노드 2->3
        nn.ReLU(),
        nn.Linear(128,10), #노드 3->4, 출력층 node 10개
        nn.Softmax()) #분류문제 -> 출력층 노드의 합을 1로 만들어줌(확률)

  def forward(self, x):#순전파
    pred=self.network(x)
    return pred

model=model()
learning_rate=0.001
optimizer=torch.optim.Adam(model.parameters(), lr=learning_rate)

torch.manual_seed(1017)
best = 100 #epoch 고정시키지 않음

for epoch in range(num_epochs): #epoch 설정
  model.train() #학습 모드
  total_train_loss=0  #사용할 변수를 미리 할당
  total_train_acc=0   #사용할 변수를 미리 할당
  for img, labels in train_loader:  #한번 돌아가면(이미지를 꺼내면) 512장씩 꺼내겠다
    img=img.reshape(-1,784) #데이터를 (장수, 길이)의 shape으로 변환

    pred=model(img)
    loss=criterion(pred,labels)

    total_train_loss += loss.item() #epoch 하나당 loss를 계산하기 위해서 batch별로 loss를 누적

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    total_train_acc += (torch.argmax(pred, dim=1)==labels).sum() / img.size(0) #argmax()->예측값C

  model.eval() #평가 모드 : 역전파가 이루어지지 않아요
  total_valid_loss=0
  total_valid_acc=0

  with torch.no_grad(): # 미분을 진행하지 않겠다 (누적되는 기울기 자체가 없다)
    for img, labels in valid_loader:
      img=img.reshape(-1,784)
      pred=model(img)

      loss=criterion(pred, labels)
      total_valid_loss+=loss.item()
      total_valid_acc += (torch.argmax(pred, dim=1)==labels).sum() / img.size(0)
```

```
train_loss=total_train_loss / len(train_loader) #loader의 길이 = 전체 데이터수 / batch_size
valid_loss=total_valid_loss / len(valid_loader)
train_acc=total_train_acc / len(train_loader)
valid_acc=total_valid_acc / len(train_loader)

print('***Epoch: {}/{}, Train Loss & Accuracy: ({:.3f,{:.3f},Valid Loss & Accuracy: ({:.3f},


if valid_loss < best: #best를 갱신
  best=valid_loss
  best_epoch=epoch+1 #best가 나온 epoch best_epoch에 저장
  print('Best Valid Loss {:.3f'.format(best_epoch.best))
  converge_cnt=0 #valid loss가 개선이 되면 0으로 초기화
else:
  converge_cnt=1 #valid loss가 개선이 되지 않으면 1씩 더해줌
if converge_cnt>2: #개선이 2 epoch이상 되지 않으면 학습을 멈추겠다
  print('Early Stopping')
      print('Best Result : Epoch {}, Valid Loss {:.3f'.format(best_epoch, best))
      break

####best valid가 안나오면 converge_cnt=1으로 갱신 -> 다음에 나오면 다시0 ####
```

CNN 기초

CNN

- MLP는 여러 hidden layer를 갖는 인공신경망을 이용해 여러 가지 문제를 해결

  but, 데이터 형상 무시
- 이미지를 MLP를 이용 → 3차원 형태의 이미지 공간적 정보 가지고 있음 , 완전연결 계층으로 데이터 넣기 위해 flatten시키면 공간적 정보 잃게됨
- 이미지 데이터 처리
  - 이미지는 3차원 형상 → 3차원으로 입력 받고 3차원으로 전달하는 신경망 사용 -CNN
  - 합성곱 신경망 (CNN) : 입력데이터에 filter 적용/필터의 윈도우를 일정 간격으로 이동
  - 편향까지 고려하여 합성곱 연산함
- CNN연산에서 고려해야할 요소들
  - 패딩(padding) : 합성곱 연산 수행 전 데이터 주변 특정값으로 채우는 것
    - 합성곱 연산 수행 후 출력 데이터의 크기가 입력 데이터와 같도록 하기 위해
    - 종류 : zero padding(0으로 채움), constant padding(상수로 채움), reflection padding(옆에 있는 값으로 채움)
  - 스트라이드(Stride) : 필터를 적용하는 위치 간격/값 클수록 출력데이터의 크기 작아짐
    - stride와 padding 사용했을 때 출력 데이터 크기 (입력 데이터:(H,W),필터 크기(F,F),패딩(P),스트라이드 (S))
    - 공식 : OH=(H+2P-F)/S +1 OW=(W+2P-F)/S + 1
  - 풀링(pooling) : 가로 세로 줄이는 연산
    - CNN은 convolutional layer 거친 후 pooling layer 거쳐 공간 크기 줄임
    - max pooling 많이 사용 → filter 크기 안에서 가장 큰 값 추출-kernel에서 가장 큰값 의미o
    - 특징 : 학습해야할 매개변수 없음, 채널 수 변하지 않음, 입력의 변화에 영향 적게 받음
- CNN layer
  - convolution/pooling : 메커니즘은 이미지를 형상으로 분할하고 분석

        - convolution : 이미지에서 특정한 패턴의 특징이 어디서 나타나는지 확인하는 도구
    - fc(fully connected layer) : 이미지를 분류/설명하는데 적합하게 예측
    - 1. feature extraction(convolution layer 모여있음) → 2.classification(fully_connected)

MNIST dataset

- 손으로 쓴 숫자들로 이루어진 대형 dataset
- 하나의 이미지 28X28의 모양으로 구성
- pytorch의 torchvision에서 제공하는 dataset → train(6000), list형태, 1X28X28의 image data,정수 label=1개

실습_LNET, CNN(fc)

```
# lenet

import torch #pytorch
import numpy as np #numpy
import matplotlib.pyplot as plt #matplotlib
import pandas as pd #pandas

#cpu
device=torch.device("cpu")

## MNIST

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

mnist_transfrom=transforms.Compose([
    transforms.ToTensor(), #Tensor형식으로 바꿔준다
    transforms.Normalize((0.5),(1.0)),

])

#train dataset, test dataset
train_dataset=MNIST(root='MNIST_data/',train=True, transform=mnist_transfrom, download=True)
test_dataset=MNIST(root='MNIST_data/',train=False, transform=mnist_transfrom, download=True)

train_dataset

#DataLoader

batch_size=128

train_loader=DataLoader(dataset=train_dataset, batch_size=128, shuffle=True)
test_loader=DataLoader(dataset=test_dataset, batch_size=128, shuffle=True)

img=train_dataset[7][0].numpy()
plt.imshow(img[0],cmap='gray')

img.shape #흑백이미지 : 1 (기본=3(RGB))

## Fully connected 모델을 만들어보자

from torch import nn
```

```
# nn.Sequential

mnist_fc_model=nn.Sequential(
    nn.Flatten(),
    nn.Linear(in_features=28*28*1,out_features=256),
    nn.Sigmoid(),
    nn.Linear(in_features=256, out_features=10),
    nn.Softmax()
)

mnist_fc_model

#device

train_dataset-> transforms.toTensor

mnist_fc_model -> to(device)

#실제학습이 일어나는 train을 만들것

from torch import optim

def train(model, epochs):

  #optimizer 설정(가중치 학습할때 어느정도 반영할지, train model의 parameter 모두 집어넣은 것)
  optimizer=optim.Adam(model.parameters(), lr=0.0001)
  #loss function
  criterion=nn.CrossEntropyLoss()  #classification 분류

  for epoch in range(epochs):
    model.train()  #model을 train 상태로
    train_accuracy=0.0

    for batch_ind, samples in enumerate(train_loader):
      x_t,y_t=samples #x는 입력 y는 출력
      #device로 x_t, y_t 보냄
      x_t,y_t=x_t.to(device),y_t.to(device)

      # x_t 넣어서 predicted
      pred=model(x_t)

      # predicted랑 y_t 비교
      # 둘의 차이 = loss 계산
      loss=criterion(pred, y_t)

      # loss값을 가지고 back-propagation weight를 업데이트
      # optimizer가 해줍니다
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

  # accuracy test for train dataset
   model.eval()  #model을 평가상태로
   correct=0
   for xx,yy in train_loader:
     data, target=xx.to(device), yy.to(device)
     pred=model(data) #pred [0,0,0.1,0.2,0.1,...0] 합 1
     _, predicted=torch.max(pred,1)
```

```
      correct+=predicted.eq(target.data).sum()

    print("train accuracy:",(100.*correct/len(train_loader.dataset)).item())


train(mnist_fc_model,10)

def get_n_params(model):
  pp=0
  for p in list(model.parameters()):
    nn=1
    for s in list(p.size()):
      nn=nn*s
    pp+=nn
  return pp

get_n_params(mnist_fc_model)
```

실습_CNN(convolution)

```
## CNN모델_convolution

mnist_cnn_model=nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3, stride=1, padding=0),
    nn.ReLU(),
    nn.Conv2d(in_channels=4, out_channels=8, kernel_size=3, stride=1, padding=0),
    nn.ReLU(),
    #CNN끝
    nn.Flatten(),
    nn.Linear(in_features=24*24*8, out_features=48),
    nn.ReLU(),
    nn.Linear(in_features=48, out_features=10),
    nn.Softmax()
)

get_n_params(mnist_cnn_model)

mnist_cnn_model

mnist_cnn_model.to(device)
train(mnist_cnn_model,10)
```

- nn.Conv2d(filter set 개수, kernal_size(filter set 사이즈),...,) → convolution함수
- 합성곱 CNN 끝나면 flatten() 1차원 배열로 만들어줌

LeNet-5

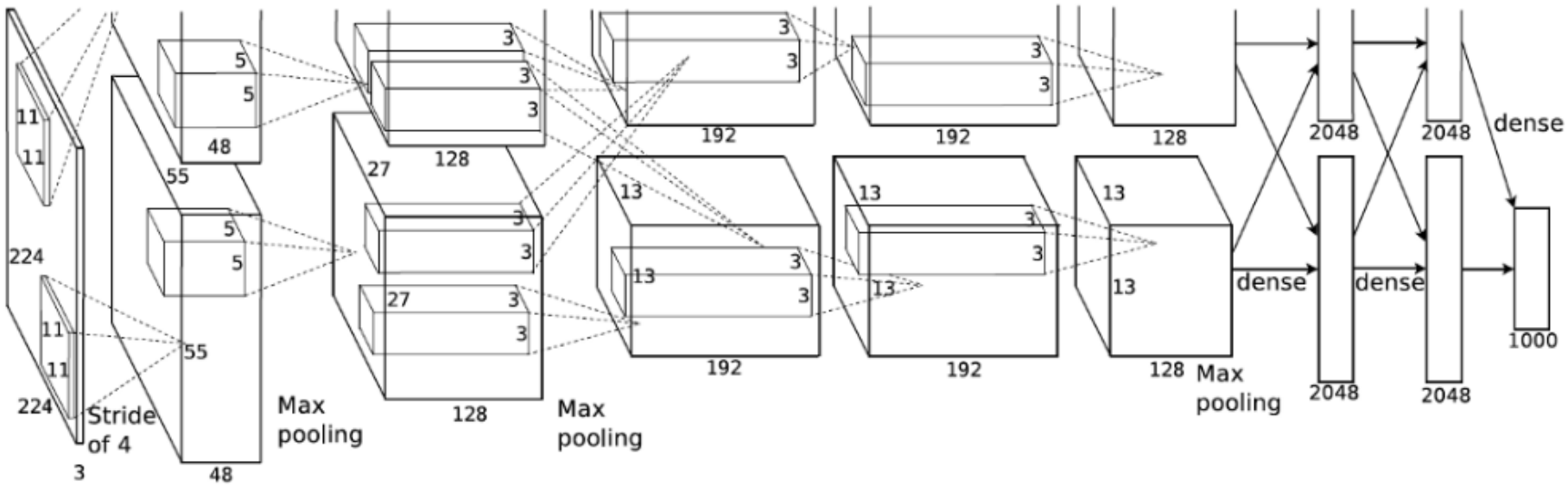
- 최초의 CNN모델
- LeNet은 Sigmoid함수 사용, 서브샘플링을 하여 중간 데이터 크기를 줄임

```
## Le-Net 5

import torch.nn.functional as F
lenet=nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1),
    nn.Tanh(), #le-net은 relu함수 없음
```

```
    nn.AvgPool2d(kernel_size=2),  #서브샘플링

    nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1),
    nn.Tanh(), #le-net은 relu함수 없음
    nn.AvgPool2d(kernel_size=2),

    nn.Conv2d(in_channels=16, out_channels=120, kernel_size=4, stride=1), #convolution -> 1X1X12
    nn.Tanh(), #le-net은 relu함수 없음

    nn.Flatten(),  #2차원자료 전결합층에 전달하기 위해 1차원으로 바꿔줌
    nn.Linear(in_features=120, out_features=84),
    nn.Tanh(),
    nn.Linear(in_features=84, out_features=10),
    nn.Softmax()
)

get_n_params(lenet)

lenet.to(device)
train(lenet,10)
```

Alexnet

- 특징
  - 기존 LeNet에 비해 훨씬 복잡한 구조
  - Max-Pooling layer 사용
  - local response normalization 사용 - 최근에는 잘 사용 x
  - gpu를 학습에 사용
    - 더 큰 네트워크를 위해 2개의 gpu를 이용해 병렬 연산 수행
    - 3번째 conv layer와 fully_connected layer를 제외하면 독립적으로 훈련 진행
    - training error줄여줌, 학습 시간 빨라짐
  - dropout 적용
    - hidden_layer의 특정 뉴런의 출력을 일정 확률로 0으로 만드는 것
    - 하나의 뉴런의 값에 크게 의존하지 않도록 모델이 학습 → 과적합 방지
  - 활성화 함수로 ReLU 사용
    - 기존 sigmoid에서는 기울기(미분값)이 0.25이므로 chain rule에 의해 계속 0.25씩 곱하면 네트워크의 초반쪽은 값 0으로 수렴하여 가중치 update가 잘 안되어 학습이 잘 안됨
  - weight decay/가중치 초기화
    - L2 regularization → 과적합 방지
    - 평균 0 표준편차 0.01의 정규분포로 가중치 초기화
    - convolution layer, fully-connected layer에 한해 1로 초기화
- 구조



- 256X256X3인 이미지를 받아서 총 1000개의 클래스로 구분해야하는 모델 → LeNet에 비해서 훨씬 복잡한 구성
- 총 2개 분기로 구성, 두 분기의 네트워크는 서로 다른 특징 학습

```
## Alexnet

import torch

device=torch.device('cuda') #gpu
torch.cuda.is_available()

from torchvision.datasets import FashionMNIST
from torchvision import transforms

fashion_mnist_transform=transforms.Compose([
    transforms.ToTensor(), #tensor형으로 변경
    transforms.Resize(224)  #크기 변경(1X28x28 -> 1X224X224)
])

download_root='./' #현재위치폴더에 다운로드

train_dataset=FashionMNIST(download_root, transform=fashion_mnist_transform, train=True, downloa
test_dataset=FashionMNIST(download_root, transform=fashion_mnist_transform, train=False, downloa

from torch.utils.data import DataLoader

batch_size=64

train_loader=DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader=DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=True)

import matplotlib.pyplot as plt

plt.imshow(train_dataset[0][0][0].numpy(), cmap='gray')

from torch import nn

model=nn.Sequential(
    #convolution
    #fully connected
    #Relu
)
class AlexNet(nn.Module):
```

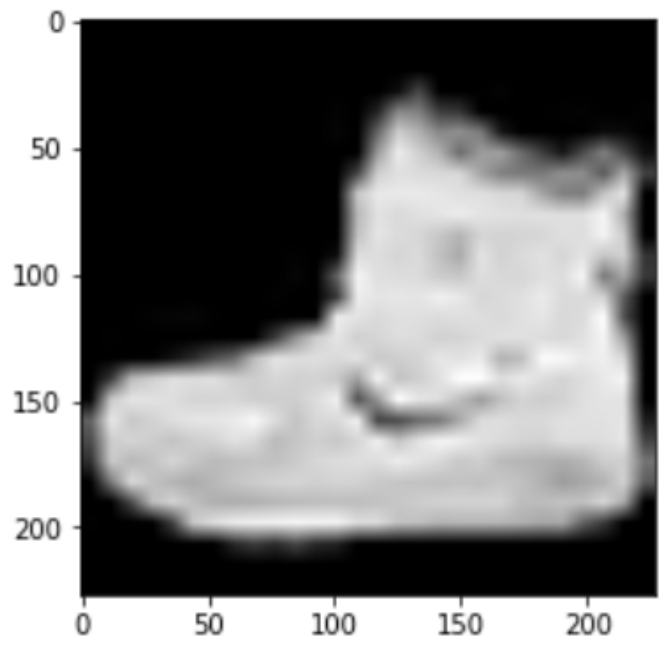
```
  def __init__(self):
    super().__init__()
    self.CNN=nn.Sequential(
        nn.Conv2d(in_channels=1, out_channels=48, kernel_size=11, stride=4, padding=0),
        nn.ReLU(inplace=True),
        nn.LocalResponseNorm(size=5,k=2),
        nn.MaxPool2d(kernel_size=3, stride=2),

        nn.Conv2d(in_channels=48, out_channels=128, kernel_size=5, stride=1, padding=2),
        nn.ReLU(inplace=True),
        nn.LocalResponseNorm(size=5,k=2),
        nn.MaxPool2d(kernel_size=3, stride=2),

        nn.Conv2d(in_channels=128, out_channels=192, kernel_size=3, stride=1, padding=1),
        nn.ReLU(inplace=True),
        nn.LocalResponseNorm(size=5,k=2),

        nn.Conv2d(in_channels=192, out_channels=192, kernel_size=3, stride=1, padding=1),
        nn.ReLU(inplace=True),
        nn.LocalResponseNorm(size=5,k=2),

        nn.Conv2d(in_channels=192, out_channels=128, kernel_size=3, stride=1, padding=1),
        nn.ReLU(inplace=True),
        nn.LocalResponseNorm(size=5,k=2),
        nn.MaxPool2d(kernel_size=3, stride=2)
    )
    self.FC=nn.Sequential(
        nn.Linear(128*6*6,2048),
        nn.ReLU(True),
        nn.Dropout(0.5), #절반의 model은 날리고 학습에 사용

        nn.Linear(2048,2048),
        nn.ReLU(True),
        nn.Dropout(0.5),

        nn.Linear(2048,10),
        nn.Softmax()
    )
    #bbbb

  def forward(self, inp):
    cnn_res=self.CNN(inp)
    #flatten 128*6*6 -> 00000
    flatten=torch.flatten(cnn_res, 1)
    fc_res=self.FC(flatten)
    return fc_res

from torch import optim

def train(model, epochs):

  #optimizer 설정(가중치 학습할때 어느정도 반영할지, train model의 parameter 모두 집어넣은 것)
  optimizer=optim.Adam(model.parameters(), lr=0.0001)
  #loss function
  criterion=nn.CrossEntropyLoss()  #classification 분류

  for epoch in range(epochs):
    model.train()  #model을 train 상태로
```

```
    train_accuracy=0.0

    for batch_ind, samples in enumerate(train_loader):
      x_t,y_t=samples #x는 입력 y는 출력
      #device로 x_t, y_t 보냄
      x_t,y_t=x_t.to(device),y_t.to(device)

      # x_t 넣어서 predicted
      pred=model(x_t)

      # predicted랑 y_t 비교
      # 둘의 차이 = loss 계산
      loss=criterion(pred, y_t)

      # loss값을 가지고 back-propagation weight를 업데이트
      # optimizer가 해줍니다
      optimizer.zero_grad()
      loss.backward()
        optimizer.step()

     # accuracy test for train dataset
      model.eval()  #model을 평가상태로
      correct=0
      for xx,yy in train_loader:
        data, target=xx.to(device), yy.to(device)
        pred=model(data) #pred [0,0,0.1,0.2,0.1,...0] 합 1
        _, predicted=torch.max(pred,1)
        correct+=predicted.eq(target.data).sum()

      print("train accuracy:",(100.*correct/len(train_loader.dataset)).item())

alexnet=AlexNet()
alexnet.to(device)
train(alexnet,10)

## train(mode,3) -> 알아서 학습

def get_n_params(model):
  pp=0
  for p in list(model.parameters()):
    nn=1
    for s in list(p.size()):
      nn=nn*s
    pp+=nn
  return pp

get_n_params(alexnet)

alexnet.eval()
correct=0
for xx,yy in test_loader:
  data, target=xx.to(device), yy.to(device)
  pred=alexnet(data) #pred [0,0,0.1,0.2,0.1,...0] 합 1
  _, predicted=torch.max(pred,1)
  correct+=predicted.eq(target.data).sum()

print("train accuracy:",(100.*correct/len(train_loader.dataset)).item())
```

```
alexnet
```

plt.imshow(train_dataset[0][0][0].numpy(), cmap='gray')

```
<matplotlib.image.AxesImage at 0x7fb0c358fb50>
```



<과정>

- Convolution layer

Conv2d → ReLU → ResponseNorm → MaxPool

- fully_Connected layer

linear → ReLU → dropout → linear 마지막 → softmax

VGG

- very deep convolutional networks for large-scale image recognition
- 더 높은 정확도를 얻기 위해 더 깊은 신경망 더 많은 layer를 갖는 CNN 등장
- 층을 깊게 할 경우 성능향상할 수 있다고 생각 → 층을 깊게 쌓는 것 → VGG
  - parameter값이 엄청 많아지는 것을 대비하여 3X3 kernal 만을 이용하여 깊이를 늘림
  - 3X3 filter가 5X5, 7X7 filter 대체함
    - 7X7 1개 = 활성화 1개 → 3X3 3개 = 활성화 3개 (활성화함수는 비선형성 증가시켜줌)
- 3X3 kernel 쓸 때 장점
  - 적은 수의 parameter : ex)3X3 3개 3*3*3=27개 < 7X7 1개 7*7*1=49개
  - 비선형성 증가 : 이러한 관점에서 1X1 filter도 비선형성 증가시킴
- 훈련 방법의 대부분은 Alexnet → batchsize=256, momentum=0.8, dropout → 첫 fc layer에 적용
  - alexnet에 비해 더 깊고 많은 paramter 가져서 모델의 복잡고 높음
  - but, 더 적은 epoch(특징 추출)
- learning rate=0.01로 시작하여 성능이 높아지지 않으면 10으로 나누어 총 3번 적용
- fc layer를 1X1 convolutional layer로 바꾸는 방법 사용
  - gpu를 이용해 훈련 진행
  - 각 batch들이 gradient(기울기)를 구하고 이들의 평균을 이용해 gradient값 사용
- 구조
  - 총 6개의 convolution network에 대해 실험진행
    - 각 network는 depth의 차이 존재

ResNet

- VGG에서 많은 레이어를 이용해 모델의 성능 높이는 작업
- 실제로는 20층 넘어가면 성능 낮아짐

- Residual learning이라는 개념을 통해 모델의 층 깊어질수록 학습 잘 되도록
- 성능 좋고, 구현 편하고, 단순함
- 문제점
  - 괒거합이 일어난 것이 아닌 레이어가 깊어질수록 정확도가 낮아짐
  - 기울기 소실/폭발도 일어남

  → 이러한 것들은 layer 깊어질수록 optimization 제대로 안돼서 생긴 문제
- Residual Block
  - H(x)를 학습하는 것보다 F(x)+x를 학습시키는 것이 더 효과적일 것이라는 전제
  - Skip connection이 핵심 → 입력 값이 일정 층을 건너뛰어 출력에 더할 수 있게함
  - Skip connection 적용 layer는 상당히 깊게 쌓아도 에러가 낮고 정확도 낮아지는 현상 발생하지 않는다.
- 구조
  - 총 34개의 layer 갖는 resnet model
  - 50층 이상의 깊은 layer를 갖는 resnet model은 bottleneck layer 이용
  - 기존 3X3 convolution layer 2개 있는 구조에서 층 하나 추가 → 1X1 layer 1개 추가
    - 연산량 줄임, layer 수 많아짐에 따라 활성화 함수 증가하여 비선형성 또한 증가

```
# Fashion MNIST

class Anet(nn.Module):
  def __init__(self):
    super().__init__()
    #CNN
    self.CNN=nn.Sequential(
      nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1, padding=1), #layer1
      nn.ReLU(inplace=True),
      nn.Conv2d(in_channels=16, out_channels=16, kernel_size=3, stride=1, padding=1), #layer2
      nn.ReLU(inplace=True),
      nn.MaxPool2d(kernel_size=2,stride=2),

      nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1), #layer3
      nn.ReLU(inplace=True),
      nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, stride=1, padding=1), #layer4
      nn.ReLU(inplace=True),
      nn.MaxPool2d(kernel_size=2,stride=2),

      nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1), #layer5
      nn.ReLU(inplace=True),
      nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1), #layer6
      nn.ReLU(inplace=True),
      nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1), #layer7
      nn.ReLU(inplace=True),
      nn.MaxPool2d(kernel_size=2,stride=2)



  )
  #FC
    self.FC=nn.Sequential(
      nn.Linear(64*29*29,256),
      nn.ReLU(True),
      nn.Dropout(0.2), #절반의 model은 날리고 학습에 사용
```

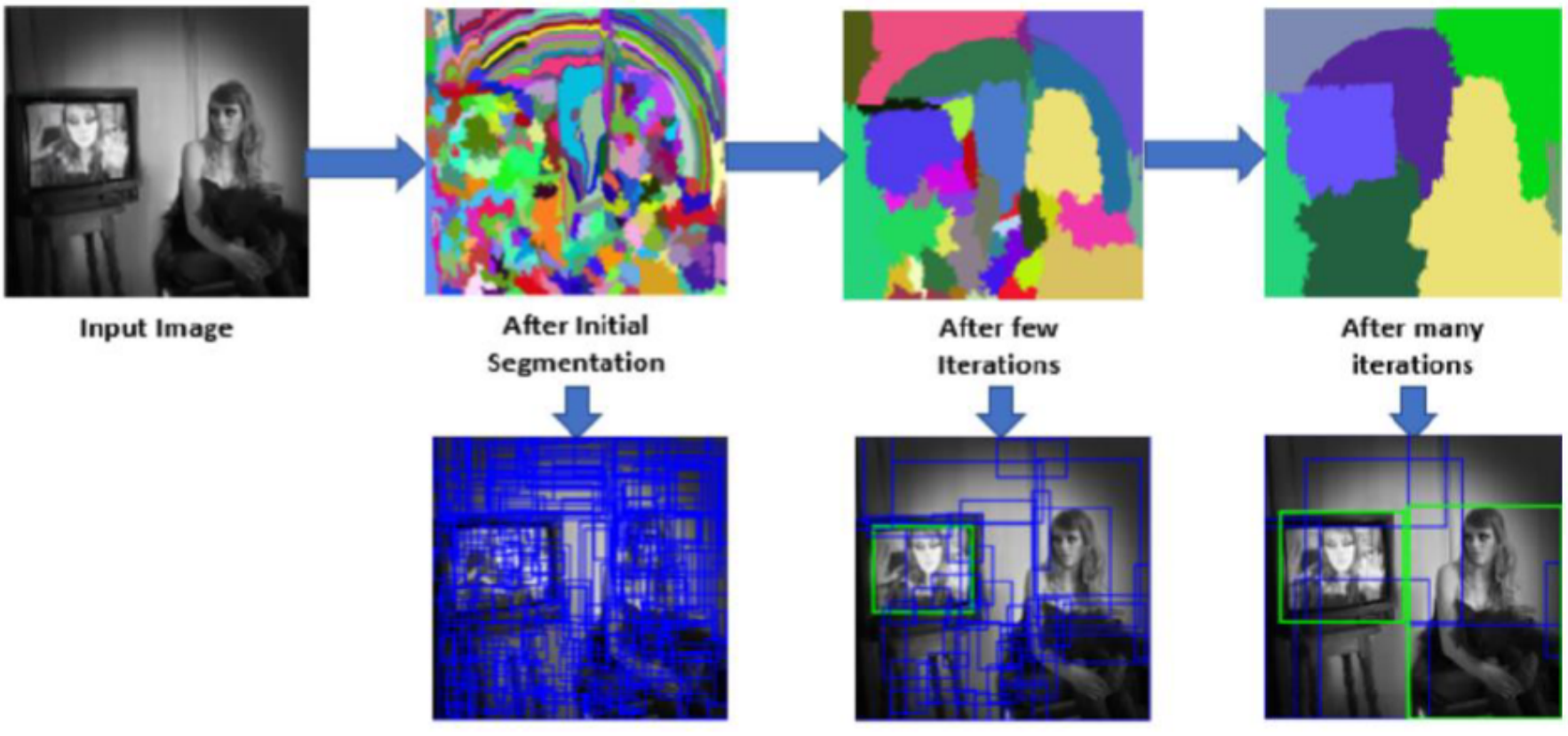
```
      nn.Linear(256,64),
      nn.ReLU(True),
      nn.Dropout(0.2),

      nn.Linear(64,10),
      nn.Softmax()
   )

     #Flatten
  def forward(self, inp):
    cnn_res=self.CNN(inp)
    flatten=torch.flatten(cnn_res,1)
    fc_res=self.FC(flatten)
    return fc_res

aaa=Anet()

get_n_params(aaa)

aaa.to(device)
train(aaa,10)
```

전이학습(Transfer Learning)

- 한 분야의 문제를 해결하기 위해 얻은 지식과 정보를 다른 문제 푸는데 사용하는 방식
- 이미지 분류와 같은 문제 해결하는데 사용했던 네트워크를 다른 데이터셋과 다른 문제에 적용시켜 푸는 것
- 이미지 분류 학습한 네트워크는 그 작업을 위해서만 학습할 때
  - 다양한 이미지의 보편적 특징 학습
  - 네트워크 깊어질수록 서로 다른 종류 특징들 학습
  - 낮은 층에서는 이미지 색, 경계 / 높은 층에서는 심화된 패턴등 학습

CNN_심화

- CNN
  - gradcam : cnn과 layer 학습하여 알아서 특징 잡아서 어떤 사물/생명인지 추측
  - 사람 얼굴 어떻게 잡아내나?
  - 특징을 말할 수 없지만 수많은 datasets들어왔을 때 대충 어떠한 사물이 무엇인지 말함 →데이터셋 클수록 정확도 높음
  - 이미지 : edge(테두리),texture(질감) 등 지역 패턴으로 분해 → 픽셀로 이루어져 수치화 가능(차이)
- dataset 클수록 좋음 → 다 찾기 어렵기 때문에 이미 존재하는 datasets을 학습시키고 내가 가지고 있는 datasets 추가하여 학습
- Detection Architecture : two stage detector(위치 예상 및 측정) → one stage detector(시간 절약, 정확도는 낮음_ex)yolo)
- Object 검출 방법
  - Region Proposal 방법 : Selective Search

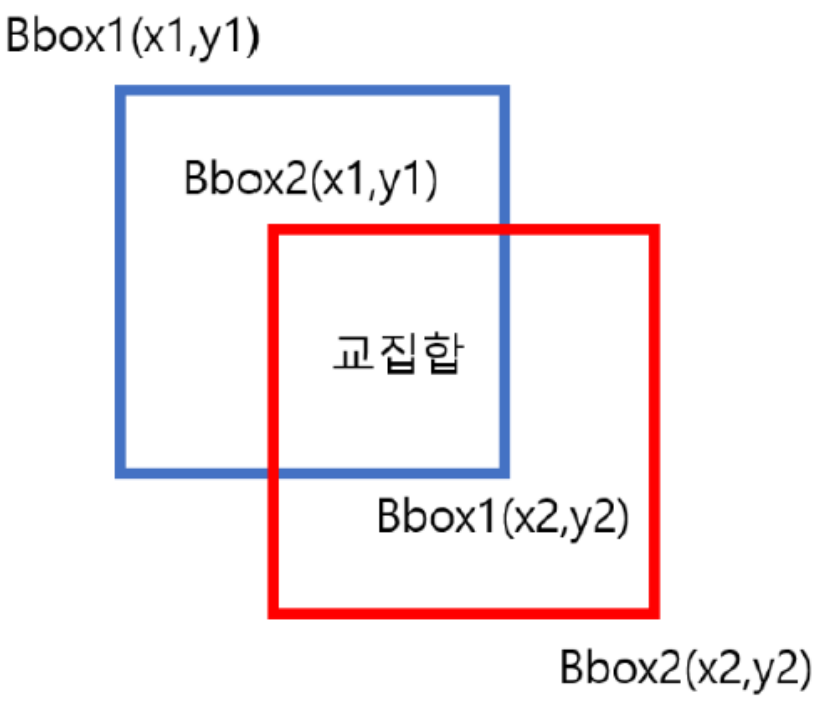


→ RGB, pixel 단위를 이미지를 변환했을 때 비슷한 것들 묶은 box가 너무 많으면 판단하기 어려움

→ 비슷한 것들끼리 묶어 유사도를 측정하여 비슷한 영역을 합친다(이때, 수식적인 부분이 있음)

→ 이렇게 합친 것들을 CNN model의 input으로 쓰임

실습_IoU



1) x1, y1, x2, y2

(x1, y1)는 좌측 상단을 의미하는 좌표

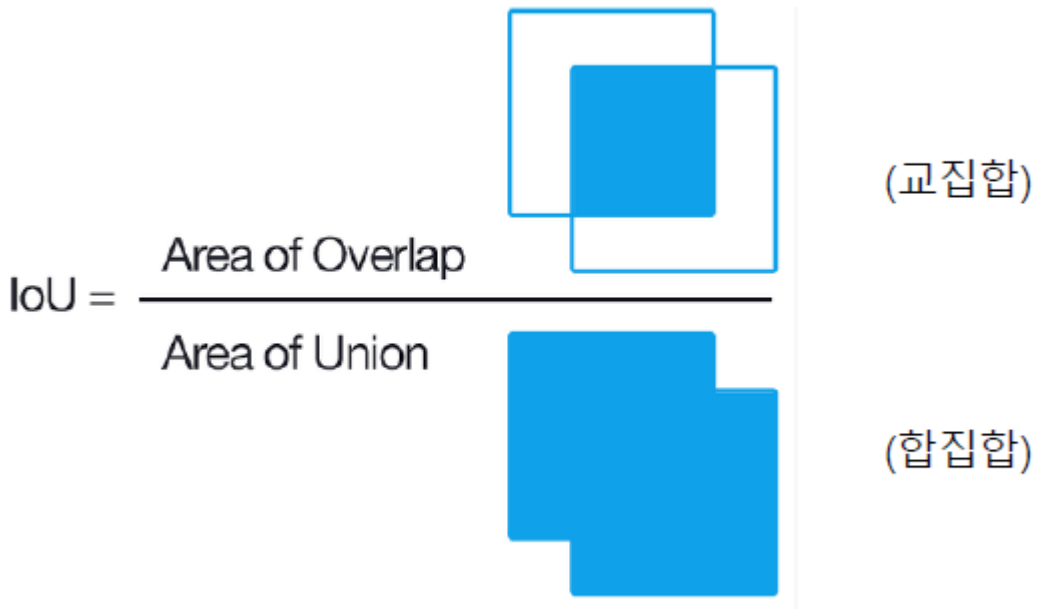
(x2, y2)는 우측 하단을 의미하는 좌표

2) x1, y1, w, h

(x1, y1)는 좌측 상단을 의미하는 좌표

w, h는 bounding box에 대한 너비와 높이

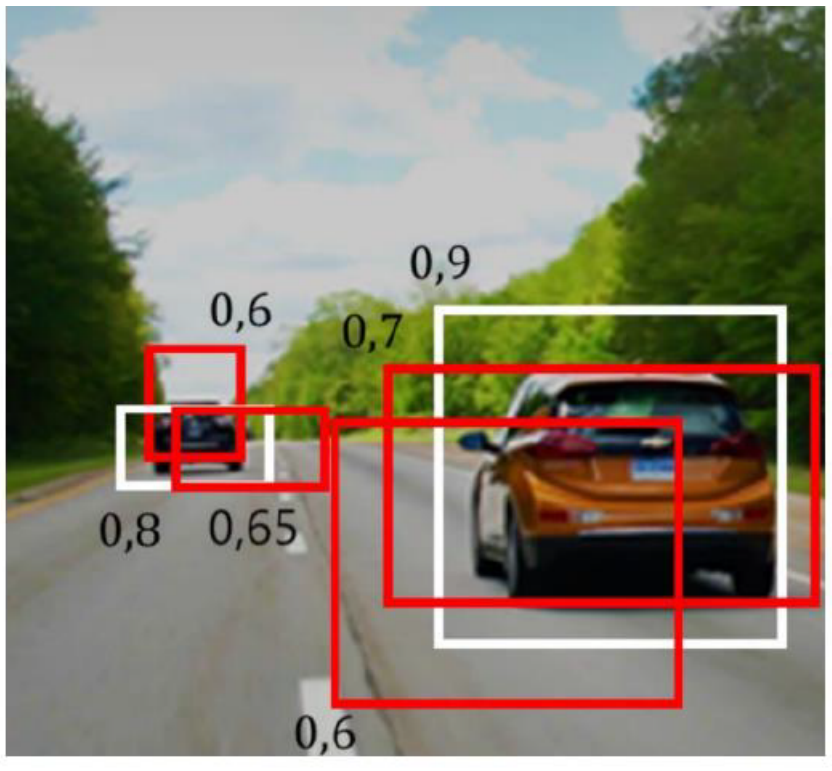
IOU : Intersection over Union : 모델이 예측한 결과와 실측(Ground Truth) Box가 얼마나 많이 겹치는가 나타내는 지표 → 클수록 정확도 높은 것




```
##IoU

def IoU(bbox1,bbox2):
  #bbox1=[x1,y1,x2,y2]
  #bbox2=[x1,y1,x2,y2]
  #교집합
  intersection_x1=max(bbox1[0],bbox2[0])  #bbox1의 x좌표,bbox2의 x좌표 비교 후 큰 큰값
  intersection_y1=max(bbox1[1],bbox2[1])  #y좌표

  intersection_x2=min(bbox1[2],bbox2[2])
  intersection_y2=min(bbox1[3],bbox2[3])

  intersection=max(intersection_x2-intersection_x1,0)*max(intersection_y2-intersection_y1,0)  #r

  #합집합
  box1_area=abs(bbox1[2]-bbox1[0])*abs(bbox1[3]-bbox1[1])
  box2_area=abs(bbox2[2]-bbox2[0])*abs(bbox2[3]-bbox2[1])

  area=box1_area+box2_area-intersection

  return intersection/area
```

실습_NMS

1. Detected 된 bounding box별로 특정 Confidence threshold 이하 bounding box는 먼저 제거(confidence score < 0.5)
2. 가장 높은 confidence score를 가진 box 순으로 내림차순 정렬하고 아래 로직을 모든 box에 순차적으로 적용.
   - 높은 confidence score를 가진 box와 겹치는 다른 box를 모두 조사하여 IOU가 특정 threshold 이상인 box를 모두 제거(예: IOU Threshold > 0.4)
3. 남아 있는 box만 선택



Confidence score가 높을 수록,

IOU Threshold가 낮을 수록 많은 Box가 제거됨.

Ground_truth_box = [x1, y1, x2, y2]

Predicted _box = [(class) , (confidence_score) , x1, y1, x2 , y2]

```
#box=[(class),(confindence_score),x1,y1,x2,y2]
threshold=0.5
bboxes=[box for box in bboxes if box[1]>threshold] #confidence_score가 0.5이상인 것들만 남김

#내림차순 정리
bboxes=sorted(bboxes,key=lambda x: x[1], reverse=True)

bboxes_after_nms=[]
iou_threshold=0.4

while bboxes:
  chose_box=bboxes.pop(0) #가장 높은 confidence_score 선택

  bboxes=[
      box
      for box in bboxes
      if box[0] != chosen_box[0] #object(class)가 같은지 비교
      or IoU(chosen_box[2:], bbox[2:])<iou_threshold #같지않다면 IoU함수 두 좌표로 계산
  ]

  bboxes_after_nms.append(chosen_box)

 return bboxes_after_nms
```

- 목적 : 최대한 필요없는 bound box 버리자.

  → NMS(Non Maximum Suppresion) : box별로 CNN에 넣음
- (정답)ground_truth_box=[x1,y1,x2,y2]
- (예측)predicted_box=[class, confidence_score,x1,x2,y1,y2] → class(어디 object), conf~(정확도)
- IoU 기본 기준 = 0.5 → 0.5보다 낮으면 다른 object로 분류

1. confidence score 높은 순으로 정렬 → object 확률 50%이하는 detect 잘 안됨 → 제거
2. 제일 높은 것과 두번째로 큰 것 비교 그 다음꺼 비교 → 더 작다고 무조건 버리면 안됨 ! 다른 object일수도 있음
3. ex) 0.9 - 0.8 비교 → IoU 작다 (다른 object일 가능성 큼) → Keep

   0.9 - 0.7 비교 → IoU 크다 → 버림

   = 가장 큰 0.9와 먼저 모두 비교하여 다른 object은 전부 keep후 0.8과 비교