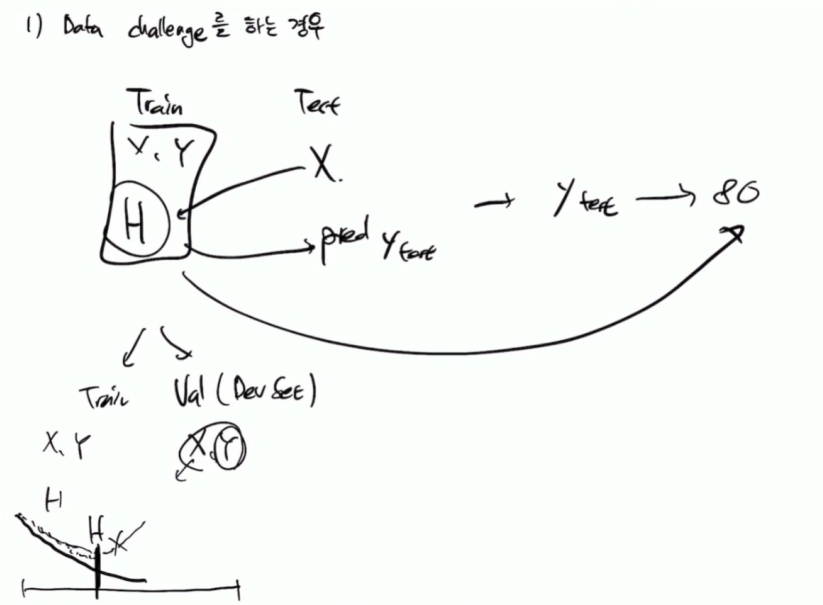
딥러닝 홀로서기#7

1. Data Challenge

- 보통 Competition에 Data Challenge가 많이 올라온다
- Train Set를 공개한 상태에서 이를 활용하여 모델을 학습시킨 후 정답이 없는 Test Set에 정답을 채워서 제출하는 문제가 나옴
- Training Set에는 X와 Y가 모두 있음, Test Set에는 X만 있음
- 우리가 만든 H(Train x, y)란 모델을 학습 시킨 후 Test에 대한 Pred Y를 도출했다면 이를 제출하여 점수를 알아볼 수 있으나, 이는 너무 느린 과정이다
- 따라서 보통 Train Set을 분할하여 Train, Validation Set(Dev Set)으로 나눈다.

  Train Set으로는 모델 H( )를 학습시키고, 나머지 Validation Set을 활용하여 Validation Loss를 측정해보면, 어느 시점에 감소 → 증가가 되는 추세를 볼 수 있다

  그럼 그 지점의 H(X)로 Test Set의 정답지를 넣어서 제출한다



2. Multi-Label Classification with Pytorch

- 방사형 데이터 분포를 가상으로 만든 후, 두 모델들이 비선형 Decision-Boundary를 형성할 수 있는지 테스트

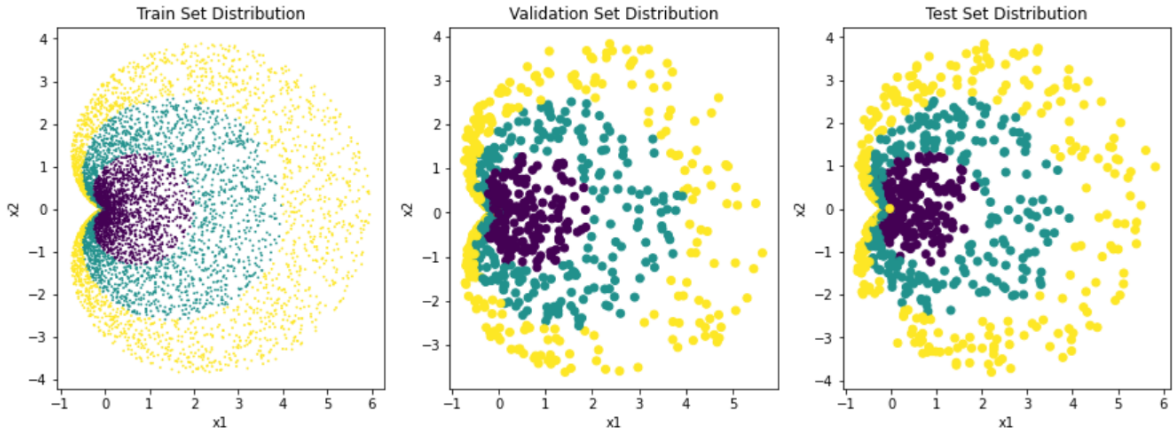
- Generating Dataset & Data Split
  - Data Set : Training Set(8000개), Validation Set(1000개), Test Set(1000개)으로 구성
  - Input (2차원) : Random Sampling X1, X2각각 10,000개씩
  - Output (3차원 (보, 초, 노)) : int형 Data Y

$$X_{train} \in \mathcal{R}^{8000\times 2}, Y_{train} \in \mathcal{Z}^{8000}$$

$X_{val} \in \mathcal{R}^{1000\times 2}, Y_{val} \in \mathcal{Z}^{1000}$

$X_{test} \in \mathcal{R}^{1000\times 2}, Y_{test} \in \mathcal{Z}^{1000}$

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

# Generating Dataset
r = np.random.rand(10000)*3
theta = np.random.rand(10000)*2*np.pi
y = r.astype(int)
r = r * (np.cos(theta) + 1)
x1 = r * np.cos(theta)
x2 = r * np.sin(theta)
x = np.array([x1, x2]).T

# Split Dataset into Train, Validation, Test
train_x, train_y = x[:8000, :], y[:8000]
val_x, val_y = x[8000:9000, :], y[8000:9000]
test_x, test_y = x[9000:,:], y[9000:]

# Visualize Each Dataset
생략
```



- Hypothesis Define (Model Define)
  - Multi-Label Logistic Model
    - Logistic Regression Model을 파이토치로 구현할 때는 Regression 문제를 풀 때와 똑같이 nn.Linear 모듈과 non-linear activation function만 달아주면 됨
    - Hypothesis : XW + b한 다음에 Softmax함수를 취한다

$$z = XW + b \ \ (W \in \mathcal{R}^{2\times 3}, b \in \mathcal{R}^{3}, z \in \mathcal{R}^{N\times 3}$$
$$H = softmax(z) \ \ (H \in \mathcal{R}^{N\times 3})$$

① W를 2차원 Input을 받아서 3차원 스페이스로 Projection한 후, Y_1, Y_2, Y_3에 b_1, b_2, b_3을 합한다

② 이후 결정된 z값을 softmax함수를 취하면 Pred Y는 Sample 수(N) x 3 행렬이 됨

```
import torch
import torch.nn as nn

class LinearModel(nn.Module):
  def __init__(self):
```

```
    super(LinearModel, self).__init__()
    self.linear = nn.Linear(in_features=2, out_features=3, bias=True)

  def forward(self, x):
    x = self.linear(x)
    return x
```

- Cost Function
  - 방식 : Cross Entropy 방식을 채택 (파이토치에 구현되어 있는 nn.CrossEntropyLoss() 함수 사용)
  - 규칙

    ⑴ Input은(test_pred_y) N x NumClass 차원으로 Float형태여야 하고,

    ⑵ Target은(test_true_y) N차원으로 각 요소는 i번째 클래스를 나타내는 int형이어야 함

    이유 : loss function은 (첫번째 파라메터)predict 값으로 각 Class의 확률값을(float) 입력 받음, (두번째 파라메터) 비교대상이 되는 실제값으로 몇 번째 Class인지를[정수형(=long)] 입력 받음 (※여기서 기본적으로 Tensor는 float형 자료를 기본적으로 만들기 때문에test_true_y1에 long()함수를 활용하여 정수형으로 변형)

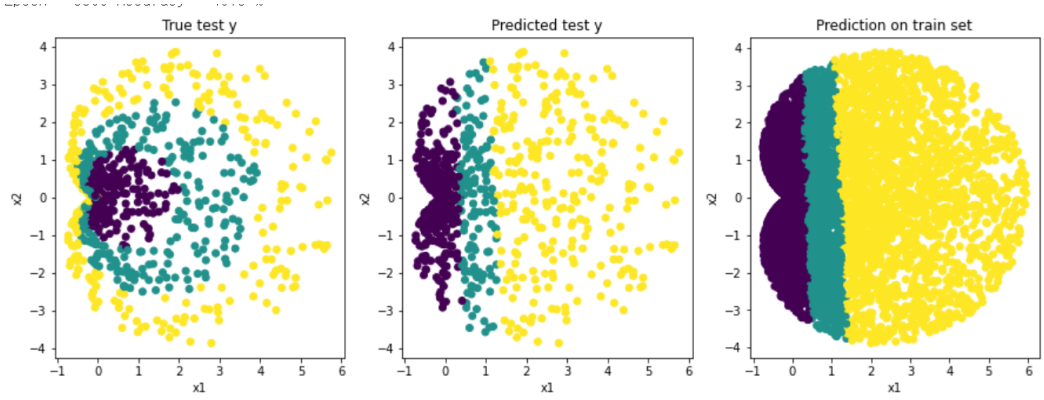
    ⑶ CrossEntropyLoss() 함수에는 Softmax가 구현되어 있으며, 함수 실행과 더불어 실행이 되기 때문에 추가적으로 Softmax 함수는 정의할 필요는 없다

```
cls_loss = nn.CrossEntropyLoss()

"""# Cross Entropy Test
test_pred_y = torch.Tensor([[1,0], [0,1]])
test_true_y1 = torch.Tensor([0,1]).long()

print(cls_loss(test_pred_y, test_true_y1))
"""
```

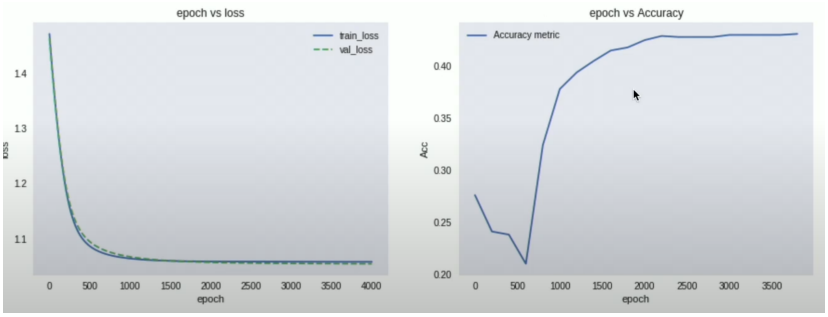
- Train & Evaluation
  - Accuracy를 활용함 (정확하게 맞춘 비율 = 정확하게 맞춘 갯수 / 모델이 만든 총 예측의 갯수)

$$Accuracy = \frac{Number\ of\ correct\ prediction}{Total\ number\ of\ predictions\ made}$$

  - 결과
    - Non-Linear데이터를 Linear Model로 분류하려다보니 완벽하게 분류가 되지 않는 모습



    - Loss가 시간이 지남에 따라 줄긴하지만, Accuracy가 50%를 채 못넘는 것을 볼 수 있다. Class가 3개인데, Random Baseline이라면 33%가 기본이기에, 실제로 그렇게 좋은 성능을 기대할 수 없다는 것을 알 수 있음




```
import torch.optim as optim
from sklearn.metrics import accuracy_score

# ====== Construct Model ====== #
model = LinearModel()

# ====== Construct Optimizer ====== #
lr = 0.005 #Learning Rate
optimizer = optim.SGD(model.parameters(), lr=lr) # Optimizer 생성

# ====== epoch값, loss 값을 저장할 리스트 생성 ====== #
list_epoch = []
list_train_loss = []
list_val_loss = []
list_acc = []
list_acc_epoch = []

epoch = 4000 # 학습 횟수(epoch)을 지정
for i in range(epoch):

# ====== Train ====== #
  model.train() # model을 train 모드로 세팅합니다. 반대로 향후 모델을 평가할 때는 eval() 모드로 변경
  optimizer.zero_grad() # optimizer에 남아있을 수도 있는 잔여 그라디언트를 0으로 다 초기화

  input_x = torch.Tensor(train_x)
  true_y = torch.Tensor(train_y).long()
  pred_y = model(input_x)
  print(input_x.shape, true_y.shape, pred_y.shape) # 각 인풋과 아웃풋의 차원을 체크

  loss = cls_loss(pred_y.squeeze(), true_y)
  loss.backward() # backward를 통해서 그라디언트를 구해줌
  optimizer.step() # step()을 통해서 그라디언트를 바탕으로 파라미터 업데이트
  list_epoch.append(i)
  list_train_loss.append(loss.detach().numpy())

# ====== Validation ====== #
  model.eval()
  optimizer.zero_grad()
  input_x = torch.Tensor(val_x)
  true_y = torch.Tensor(val_y).long()
  pred_y = model(input_x)
  loss = cls_loss(pred_y.squeeze(), true_y)
  list_val_loss.append(loss.detach().numpy())

# ====== Evaluation ======= #
  if i % 200 == 0: #200회의 학습마다 실제 데이터 분포와 모델이 예측한 분포를 그림
    # ====== Calculate Accuracy ====== #
    model.eval()
    optimizer.zero_grad()
    input_x = torch.Tensor(test_X)
    true_y = torch.Tensor(test_y)
    pred_y = model(input_x).detach().max(dim=1)[1].numpy()
    acc = accuracy_score(true_y, pred_y) # sklearn 쪽 함수들은 true_y 가 먼저, pred_y가 나중에 인자로 들어가는 것에 주의합시다
    list_acc.append(acc)
    list_acc_epoch.append(i)

    fig = plt.figure(figsize=(15,5))

    그래프 코드는 생략
```