Linear classifier

이재형

국민대학교 전자공학부

harry8121@kookmin.ac.kr

요 약

Linear Classifier의 지도학습 기법을 간단한 2,3 class의 toy dataset을 통하여 구현해보았습니다.

I. 서론

Linear Classifier의 지도학습 기법을 구현하는 전 과정에 대하여 상세한 서술을 해보겠습니다.

코드 구현 순서에 맞추어 'Toy data 생성' -> 'Toy datset train/test set 분할' -> 'Linear Classifer모델' -> 'Softmax기반 손실함수' -> 'Linaer Classifier모델 학습' 순서로 서술하겠습니다.

특정 함수에 대해 서술할 때, 매개변수에 실제값을 대입하여 특정 이미지의 크기나 수에 대해서 직관적인 파악이 가능하도록 서술하는 경우가 있음을 미리 말씀드리는 바입니다.

II. 수행 내용

2.1 Toy data 생성

2.1.1 이미지 생성

흑/백 밝기를 갖는 이미지와 백/흑 밝기를 갖는 이미지를 각각 100개씩 생성하였습니다.

- class 0 : 흑/백, class 1 : 백/흑
- 이미지의 크기 : 128x128

- 128x128 이미지가 저장될 0으로 구성된 배열 생성
  > db_imgs = np.zeros(200, 128, 128)

- class 0 : db_imgs[: 100, : , 64 :] = 1
- class 1 : db_imgs[100 : 200, : , : 64] = 1

```
# 이미지의 중간 열 인덱스 -> 클래스간의 구분을 위해 사용
mid_idx = int(img_sz/num_cls)
# class 0: black/white - # 흑/백 밝기를 갖는 영상 생성
db_imgs[:num_img_per_cls,:,mid_idx:] = 1
# class 1: white/black - # 백/흑 밝기를 갖는 영상 생성
db_imgs[num_img_per_cls:2*num_img_per_cls, :, :mid_idx] =
```

이미지 생성 코드

2.1.2 data, label 생성

생성한 이미지로부터 data와 class label을 생성하였습니다.

- flatten to 1D vector
  > db_data = db_imgs.reshape(200, -1)

- class label 생성
  > db_labels = np.zeros(200)

- class 0 : db_labels[: 100] = 0
- class 1 : db_labels[100 : 200] = 1

```
# flatten each image to 1D vector
# 총영상개수x128x128 영상 배열을 총영상개수x128^2 배열로
db_data = db_imgs.reshape(num_imgs, -1)
```

```
# class label 생성
db_labels = np.zeros((num_cls*num_img_per_cls),dtyp
db_labels[:num_img_per_cls] = 0
db_labels[num_img_per_cls:2*num_img_per_cls] = 1
```

2.2 Toy dataset - training/test 분할

2.2.1 idxs 설정

이미지를 random하게 섞기 위해서 index 배열 생성하였습니다. 각 클래스 별로 100개의 이미지가 존재하기에 index배열의 arange는 100입니다.

- index 배열 생성
  > idxs = np.arange(100)

- random shuffle
  > np.random.shuffle(idxs)

2.2.2 train data 확보

전체 데이터 중 20%의 비율로 설정하여 train data를 확보하였습니다. 위에서 random성을 부여해 주는 장치 역할을 하는 idxs를 이용하여 인덱싱하여 전체 db_data로부터 train_data를 확보하였습니다
또한 train data의 label data또한 idxs를 이용하여 인덱싱을 통해 확보하였습니다.

- 전체 데이터 중 20% 비율
 > num_tr_data = 100 x 0.2 = 20개(class당)

- train data의 수만큼 행을 갖고, 각 행은 이미지 데이터를 표현하는 1D vector로 구성된 tr_data생성
 > tr_data = np.zeros(20x2, 128^2)

- 각 class별 idxs를 통한 인덱싱으로 train data확보
 > tr_data[: 20, :] = db_data[idxs[:20], :]
 > tr_data[20:40, :] = db_data[100+idxs[:20], :]

- train data의 label data 생성
 > tr_labels = np.zeros(20x2, dtype='int8')

- idxs를 통한 인덱싱으로 train label data확보

```
tr_labels = np.zeros((num_tr_data*num_cls),dtype='int8')
# 학습용 레이블 데이터 확보
tr_labels[:num_tr_data] = db_labels[idxs[:num_tr_data]]
tr_labels[num_tr_data:2*num_tr_data] = db_labels[num_img_per_cls+idxs[:num_tr_dat
```

train_label data 확보

2.2.3 test data 확보

전체 데이터 수에서 train data수를 제외한 나머지 데이터를 test data로 확보하였습니다. 즉, class당 100 - 20 = 80개의 test data를 확보한 것입니다.

test data를 생성하고 얻는 과정과 test_label data를 생성하고 얻는 과정은 train data와 동일한 방식으로 진행되었기에 생략하도록 하겠습니다.

2.3 Linear Classifier 모델 생성

이제 데이터를 생성하고, train/test set을 확보했습니다. 이 데이터를 활용할 모델을 생성하는 과정에 대해 서술하겠습니다.

기본적으로 Linear Classifier는 들어온 입력 데이터와 모델의 파라미터 w의 내적 연산과 편향성을 고려하여 입력 데이터의 클래스 점수를 계산해야 합니다. 이 과정이 forward함수에 해당됩니다.

또한, 그렇게 도출된 score를 정답 score와 비교하고 손실을 계산해서 결국에는 가장 데이터를 잘 표현하는 파라미터를 찾는 것이 우리의 목표입니다. 그렇게 하기 위하여 역전파 계산과 경사하강법은 필수적입니다. 위의 3가지가 각각의 함수로 어떻게 구현되었는지 서술하겠습니다.

2.3.1 forward 연산

우선적으로 고려할 것이 데이터의 차원과 클래스의 수입니다. 현재 data의 차원은 NxD이고, 클래스의 수는 M이라고 가정하겠습니다,

- 파라미터 w와 bias값 b를 random하게 작은 값으로 초기화

- W의 차원 : DxM
 > self.W = np.random.randn(data_dim, num_cls)

- b의 차원 : M,
 > self.b = np.random.randint(num_cls)

- 입력 데이터 x(NxD) * 파라미터 W(DxM) + bias(M,)
 > self.scores = np.dot(x, w) + b

```
def forward(self, data):
    self.data = data
    self.scores = np.dot(data, self.W) + s
    # data dimension is N x D
    return self.scores
```

foward 연산

2.3.2 backward 연산

입력받는 upstream gradien를 통해 역전파 연산을 수행하는 함수입니다. 파라미터 W와 편향 b에 대한 gradient를 계산합니다.

- upstream gradient의 차원 : NxM
- data의 차원 : NxD -> data.T : DxN
 > grad_W = np.dot(self.data.T, upstream_grad)

- b의 gradient : 상류 gradient의 열방향 합
 > grad_b = np.sum(upstream_grad, axis=0)

2.3.3 경사하강법

backward연산에서 계산된 gradient를 통해서 경사하강법을 적용했습니다. 이를 통해 모델의 파라미터를 업데이트 하는 역할을 수행하는 함수를 작성하였습니다.

파라미터 w와 편향값 b 각각 learning rate를 고려한 경사하강법 식을 구현하였습니다.

- self.W = self.W - lr * grad_W
- self.b = self.b - lr * grad_ b

```
def grad_descent(self, grad, lr):
    grad_W = grad['W']
    grad_b = grad['b']
    self.W = self.W - lr*grad_W
    self.b = self.b - lr*grad_b
```

경사하강법

2.4 Softmax Loss function

softmax 함수 기반의 cross-entropy loss함수를 구현하고, 모델을 학습하는데 필요한 gradent를 계산하는 과정을 포함한 SoftMaxLoss 클래스에 대해서 서술하겠습니다.

2.4.1 Softmax 클래스 정의

- 생성자 : __init__(self, model, num_cls)
- self.model_scores, self.softmax_probs, self.loss, self.labels 등을 초기화

```
def __init__(self, model, num_cls):
    # set model object
    self.model = model
    self.model_scores = 0
    self.softmax_probs = 0
    self.loss = 0
    self.labels = 0
    self.data = 0
    self.probs = 0
    self.num_cls = num_cls
```

Softmax class 생성자

2.4.2 Softmax 확률 계산

- softmax 확률 계산 함수 정의
  > def compute_probs(self, data) :

- data를 입력받아서 softmax 확률을 계산

$$>P(Y=k \mid X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

1. score 계산

- scores = self.model.forward(data)
- 'forward' 메서드에 data를 전달하여 그 data가 각 클래스에 속할 확률을 계산

2. softmax 함수 계산

- exp_scores = np.exp(scores - np.max(scores, axis=1, keepdims=True)
- 위의 식은 softmax함수식의 분자 부분
- 특이점이 있다면 각 점수에서 최대 점수를 빼준다는 것입니다.
- 지수함수는 입력값이 커짐에 따라 무한히 커지기 때문에 수치적 불안성이 존재합니다. 이를 방지하고 수치적 안정성을 위해 scores 행렬의 각 행마다 최대 점수를 빼줍니다.
- 최대값을 빼주어도 확률간의 상대적 크기와 순서는 동일하게 유지되어 softmax함수의 출력값도 변하지 않습니다.

- softmax_probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
- 위는 softmax함수의 전체출력을 나타낸 식입니다. 분모에 들어가는 수식은 각 data에 대한 모든 클래스의 지수값을 합산한 것으로 이는 정규화 상수의 역할을 합니다.

- self.probs = softmax_probs
- loss, gradient계산에 활용될 확률값을 저장합니다.

```
def compute_probs(self, data):
    scores = self.model.forward(data)
    self.model_scores = scores
    # compute softmax function
    exp_scores = np.exp(scores - np.max(scores, axis=1, keepdims=True))
    softmax_probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=Tr
    self.probs = softmax_probs
```

softmax 확률 계산 함수

2.4.3 Softmax Loss 계산

- def compute_loss(self, data, labels)
- 데이터와 해당 데이터의 정답 레이블을 입력으로 받아 loss를 계산

1. data 입력 & 확률 계산

- self.data = data
- self.compute_probs(data)

2. Loss 계산

- num_samples = data.shape[0]
- 입력 데이터의 행수를 측정하여 데이터수 저장

-correct_class_prob = self.probs[np.arange(num_samples), labels]
- 각 데이터 포인트에 대한 정답 클래스의 확률을 가져옴
- np.arange(num_samples) : 0 ~ num_samples -1 까지의 배열
- labels : 각 데이터 포인트에 대한 실제 정답 클래스 레이블을 갖고 있는 변수

- loss = -np.log(correct_class_prob).sum() / num_samples
- 정답 클래스의 확률에 대한 음의 log값을 취하는 cross-entropy loss를 적용합니다.
- 그 후에 모든 데이터 포인트에 대한 손실을 합산하고, 데이터 수만큼 나누어서 평균 loss를 구합니다.

위의 모든 과정은 softmax 함수를 통해 예측 확률을 얻고, cross-entropy loss를 적용하여 모델의 예측과 실제 레이블 간의 차이를 측정하는 과정인 것입니다.

```
def compute_loss(self, data, labels):
    self.data = data
    self.compute_probs(data)

    num_samples = data.shape[0]
    correct_class_prob = self.probs[np.arange(num_samples), lab
    loss = -np.log(correct_class_prob).sum() / num_samples
    self.loss = loss
```

softmax loss 계산 함수

2.4.4 Softmax Loss의 gradient 계산

- def compute_grad(self, labels)
- softmax loss에 대한 모델 파라미터의 gradient를 계산

1. softmax 확률 저장

- z = self.probs

- 데이터 포인트 수(N) x클래스 수(M)의 형태로 확률이 저장됩니다.

2.grad_z = z - make_labels_onehot(self.labels,self.num_cls)

- softmax확률 z에서 정답 클래스에 대한 '원-핫 인코딩'을 빼줍니다.
- 정답 클래스의 레이블을 '원-핫 인코딩' 형태로 변환하면 정답클래스에 대한 위치만 1로 표시되고 나머지는 0이 됩니다.

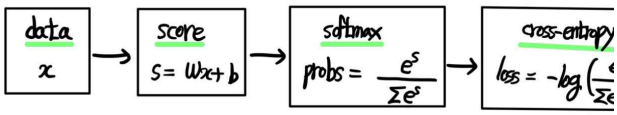
- 즉, 각 데이터 포인트에 대해 모든 클래스에 대한 예측 확률과 정답클래스에 대한 원-핫 vector의 차이를 계산하는 것입니다. 이는 loss 함수의 gradient연산에 이용됩니다.

3. self.model.backward(grad_z)

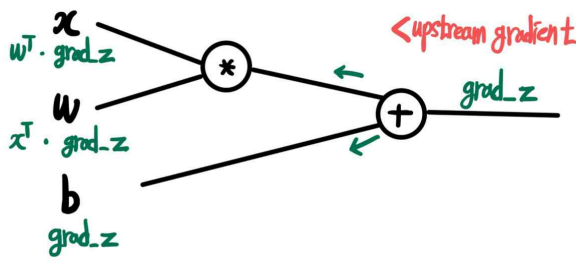
- 모델의 backward 메서드를 호출하여 역전파 알고리즘을 통해 gradient를 계산하고자 하는 것입니다.
- grad_z는 loss 함수에 대한 gradient를 의미하고 이를 통해서 모델의 파라미터를 업데이트 합니다.

```
def compute_grad(self, labels):
    # score dimensions should be N x M
    z = self.probs
    # grad_z dimensions should also be N x M
    grad_z = z - make_labels_onehot(self.labels,self.n
    return self.model.backward(grad_z)
```

softmax loss의 gradient 계산 함수



foward



gradient computational graph

2.5 Linear Classifier 학습

위의 모든 과정을 통해 데이터를 생성하고, 분할하였고, 모델 객체를 생성하고, 손실함수 또한 정의하였습니다. 이제 mini-batch data를 추출하고 data를 통해 모델을 학습하는 함수를 구현하여 학습시키는 부분입니다.

2.5.1 mini-batch 생성

- def get_mini_batch(batch_sz, db_data,
db_labels, num_data) :

- 주어진 데이터에서 'batch_sz'만큼 데이터 포인트와 해당 레이블을 무작위로 추출하여 반환합니다.
- 이때, train/test data를 무작위로 분할하기 위해 사용했었던 idxs인덱스를 동일한 방법으로 사용합니다.

```
def get_mini_batch(batch_sz, db_data, db_labels, num_data):
    num_samples = db_data.shape[0]
    idxs = np.arange(num_samples)
    np.random.shuffle(idxs)  # 데이터 인덱스를 무작위로 섞음

    batch_data = db_data[idxs[:batch_sz]]  # 미니 배치 데이터 추출
    batch_labels = db_labels[idxs[:batch_sz]]  # 미니 배치 레이블 추

    # VisualizeBatchData(batch_data,batch_sz)
    return batch_data, batch_labels
```

mini-batch 생성 함수

2.5.2 Linear Classifier train 함수

- def train_linear_classifier(db_data, db_labels, num_cls,
lr, batch_sz, num_iter) :

```
# define & initialize linear classifier
num_data = db_data.shape[0]
data_dim = db_data.shape[1]
model = LinearClassifier(data_dim, num_cl
# define loss function
loss_fn = SoftMaxLoss(model,num_cls)
```

- data의 수, data의 차원을 파악하고 이를 고려하여 모델을 생성하고, 손실함수를 초기화합니다.

```
# iterative stochastic gradient descent
for i in range(num_iter):
    # sample data mini-batch
    batch_data, batch_labels = get_mini_batch(batch_sz, db_data, db_labels
    model.data = batch_data
    loss_fn.labels = batch_labels
    # compute loss value
    loss = loss_fn.compute_loss(batch_data, batch_labels)
    # print('loss=',loss)
    # compute gradient (backpropagation)
    grad = loss_fn.compute_grad(batch_labels)
    # update model parameters using gradient
    model.grad_descent(grad, lr)
model.visualize_weights()
return model
```

- 주어진 에포크 수 num_iter동안 학습을 반복합니다
- SGD를 위해서 batch_data, batch_label에 크기가 'batch_sz'인 미니 배치 데이터와 해당 레이블을 무작위로 추출합니다.

- model.data = batch_data
- loss_fn.labels = batch_labels
- 위의 2줄은 미니 배치 데이터를 모델 객체와 손실 함수 객체에 직접 지정한 코드입니다.
- label값이 모델로 전달되지 않는 오류를 극복할 수 있었습니다.

- 그 후에는 loss를 계산하고, gradient를 계산한 후에 경사하강법을 이용하여 최적의 모델 파라미터를 찾는 과정을 반복합니다.

3. 실험 결과 및 분석

이렇게 구현된 Linear Classifier를 통해 test data의 판별 결과를 도출해보았습니다.

- te_scores = model.forward(te_data)
- te_predictions = np.argmax(te_scores, axis=1)
- test data의 score를 계산하고, 판별 결과를 가장 높은 score를 갖는 클래스로 선택하여 변환하였습니다.

- correct_predictions = (te_predictions == te_labels)
- 예측값과 실제 레이블이 일치하는 경우 True

- Acc = np.mean(correct_predictions.astype(float))
- print('Classification accuracy= {}%'.format(Acc*100))
- 정확한 예측의 비율을 계산하여 백분율로 환산하였습니다.

```
te_scores = model.forward(te_data)
# 판별 결과를 클래스로 변환 (가장 높은 점수를 갖는 클래스 선
te_predictions = np.argmax(te_scores, axis=1)

# 정확도 계산
# 예측값과 실제 레이블이 일치하는 경우 True
correct_predictions = (te_predictions == te_labels)
# 정확한 예측의 비율 계산
Acc = np.mean(correct_predictions.astype(float))

print('Classification accuracy= {}%'.format(Acc*100))
```

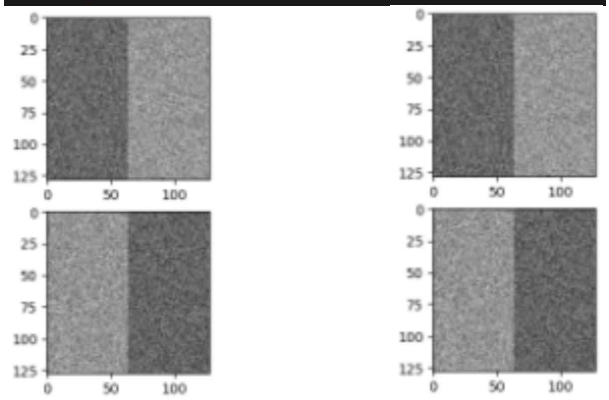
test data classification

```
Classification accuracy= 100.0
```

예측 정확도

간단한 2class로 구성된 toy dataset을 이용하였기에 정확도가 100%임을 확인할 수 있었습니다.

```
# if i == 1 :
#     model.visualize_weights()
# 1epoch 후에 파라미터를 시각화
```



위의 왼쪽 사진은 모델을 100 epoch로 학습할 때 for문이 1번 돌고나서의 파라미터를 시각화한 것이고, 오른쪽은 for문이 종료되고 나서 즉, 학습이 종료된 후에 파라미터를 시각화한 것입니다. 이러한 결과가 나오는 이유를 분석해보았습니다.

1. 데이터 크기와 복잡성

- 데이터 셋이 작고 간단한 경우, 모델은 데이터를 빠르게 학습하고 수렴할 수 있게 됩니다. 따라서 더 많은 epoch을 추가로 수행해도 모델의 파라미터가 크게 변하지 않을 수 있습니다.

2. 모델의 복잡성

- 사용한 Linear Classifier모델이 간단하고 파라미터 수가 적은 경우, 마찬가지로 더 많은 epoch을 수행해도 모델이 빨리 수렴하고 파라미터가 크게 변하지 않을 수 있습니다.

3. 학습률(Learning rate)

- 학습률이 적절하게 설정이 되었을 때, 모델은 빠르게 수렴하고 더 빨리 학습을 마칠 수 있습니다.

위와 같은 이유로 분석해보았습니다. 위의 과제에서 사용한 모델은 데이터의 단순함과 간단한 모델을 사용하였기에 충분히 나올 수 있는 결과임을 인지할 수 있었습니다.

4. 3class toy dataset(별첨)

위의 모든 과정을 2 class가 아닌 3 class toy data를 생성하여 진행해보았습니다. 흑/백, 백/흑 밝기를 갖는 2가지 class의 영상에서 영상을 3등분하여 3가지 class를 갖는 data를 생성하였습니다.

- class 0 : 백/흑/흑
- class 1: 흑/백/흑
- class 2: 흑/흑/백

위와 같이 3가지의 class를 갖는 data를 생성하여 classification을 진행하였습니다. 데이터 생성과 전처리, train/test 분할 과정에서 달라진 부분을 중점적으로 서술하겠습니다.

4.1 data 생성 & 전처리

3가지 클래스의 toy data를 생성할 때, 위에서 진행한 2가지 클래스인 toy data와의 다른 부분들에 대하여 서술하겠습니다.

- 이미지 크기와 각 클래스 당 이미지 수는 동일
- num_cls = 3으로 변경

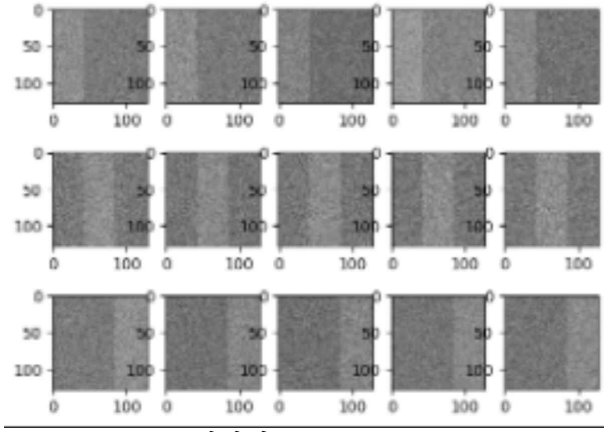
- db_imgs[:100, : , :42] = 1
- db_imgs[100:200, : , 42: 2x42] = 1
- db_imgs[200: , : , 2x42:] = 1

- 3 class이므로 변경된 mid_idx = 42를 기준으로 각 클래스 당 100개의 이미지를 생성
- 위에서 제시한 class 0,1,2 각각의 밝기 조건을 만족하도록 인덱싱

- db_labels = np.zeros(3*100, dtype='int8')
- db_labels[:100] = 0
- db_labels[100:200] = 1
- db_labels[200:] = 2
- 3가지 클래스에 대한 라벨링 또한 진행

```
# 클래스 0: 백/흑/흑
db_imgs[:num_img_per_cls, :, :mid_idx] = 1
# 클래스 1: 흑/백/흑
db_imgs[num_img_per_cls:2*num_img_per_cls, :, mid_idx:2*mid_idx] =
# 클래스 2: 흑/흑/백
db_imgs[2*num_img_per_cls:, :, 2*mid_idx:] = 1
```

```
db_labels = np.zeros((num_cls * num_img_per_cls), dtype='int
db_labels[:num_img_per_cls] = 0
db_labels[num_img_per_cls:2 * num_img_per_cls] = 1
db_labels[2 * num_img_per_cls:] = 2
```



생성된 3class toy data

4.2 toy dataset train/test 분할

생성된 3class toy data를 train과 test data로 분할하는 과정을 진행하였습니다. 2class에서 진행한 train data의 비율은 0.2와 동일하고, test data또한 전체 데이터에서 train data를 제외한 나머지로 할당하였습니다.

- for i in range(num_cls) :
  tr_data[i * num_tr_data: (i+1) * num_tr_data, :] =
  db_data[i * num_img_per_cls + idxs[:num_tr_data], :]

- for i in range(num_cls) :
  tr_labels[i * num_tr_data : (i+1) * num_tr_data] =
  db_labels[i * num_img_per_cls + idxs[:num_tr_data]]

- train/test data와 해당 레이블을 생성할 때, 2 class toydata와 다르게 for문을 통해서 iterative하게 코드를 작성하였습니다.

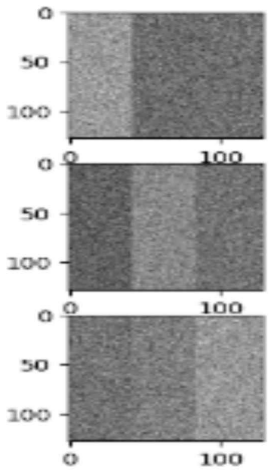
```
# 훈련 데이터 생성
for i in range(num_cls):
    tr_data[i * num_tr_data:(i + 1) * num_tr_data, :] = db_data[i * num_img_per_cls + idxs[:num
# 훈련 데이터 레이블 초기화
tr_labels = np.zeros((num_tr_data * num_cls), dtype='int8')
# 훈련 데이터 레이블 생성
for i in range(num_cls):
    tr_labels[i * num_tr_data:(i + 1) * num_tr_data] = db_labels[i * num_img_per_cls + idxs[:n
```

train data

```
# 테스트 데이터 생성
for i in range(num_cls):
    te_data[i * num_te_data:(i + 1) * num_te_data, :] = db_data[i * num_img_per_cls + idxs[num_tr_data:num_
# 테스트 데이터 레이블 초기화
te_labels = np.zeros((num_te_data * num_cls), dtype='int8')
# 테스트 데이터 레이블 생성
for i in range(num_cls):
    te_labels[i * num_te_data:(i + 1) * num_te_data] = db_labels[i * num_img_per_cls + idxs[num_tr_data:num
```

test data

나머지의 전과정은 2class toy data와 동일하게 진행되었습니다.




```
Classification accuracy= 100.0%
```

모델 학습 종류 후의 파라미터는 위와 같고, 모델의 예측값과 실제값이 100% 일치함을 확인할 수 있었습니다.

5. 결론

간단한 데이터지만, toy data를 numpy를 이용하여 직접 생성하고 라벨링해보며 데이터 전처리 과정에 조금 더 익숙해질 수 있었습니다.

또한, Linear Classifier 모델을 구현하며 모델이 처리하는 연산의 과정에 대해 완벽하게 숙지할 수 있었으며 결국 머신러닝의 최종 목표인 모델을 가장 잘 나타내는 파라미터 W를 찾는 것을 다시 한 번 되새길 수 있었습니다.

참고문헌

[1] chat-gpt : 데이터와 모델의 복잡성에 따른 파라미터의 변화에 대한 내용을 gpt로부터 얻었습니다.
[2] 강의자료 - Linear Classifier & Linear Regression