# Chapter 4 Part 4 요약 (Page 163-171) 091520 강인승

## The MNIST Loss Function

- 이미지 파일 하나가 각각 independent variable 임
  - 모든 이미지를 다 stack 해서 하나의 랭크 3 의 텐서로 만든다 (랭크 2 의 탠서를 가지는 리스트)
    - View 라는 파이토치의 함수를 이용
    - View 함수의 -1 파라미터는 데이터 사이트 axis 를 늘려서 여러 탠서가 합쳐질 수 있게함

```
train_x = torch.cat([stacked_threes, stacked_sevens]).view(-1, 28*28)
```

    - Stack 된 데이터에 라벨링을 함

```
train_y = tensor([1]*len(threes) + [0]*len(sevens)).unsqueeze(1)
train_x.shape,train_y.shape
```

```
(torch.Size([12396, 784]), torch.Size([12396, 1]))
```

    - Zip 함수를 이용해서 데이터 셋을 만든다

```
dset = list(zip(train_x,train_y))
x,y = dset[0]
x.shape,y
```

- 모델 학습의 7 단계에서 우선 1 단계인 weight 을 초기화 한다

```
def init_params(size, std=1.0): return (torch.randn(size)*std).requires_grad_()
```

```
weights = init_params((28*28,1))
```

- 앞에서 예기한 weights * pixel 은 적합하지 않다, pixel 값이 0 이면 결과값이 0 이 되어버림
  - 파라미터는 weight 과 bias 를 포함
  - 하나 이미지의 prediction step 의 예

```
(train_x[0]*weights.T).sum() + bias
```

```
tensor([20.2336], grad_fn=<AddBackward0>)
```

  - 이 방식으로 training data X 를 한번에 예측값을 계산함

```
def linear1(xb): return xb@weights + bias
preds = linear1(train_x)
preds
```

```
tensor([[20.2336],
        [17.0644],
        [15.2384],
        ...,
        [18.3804],
        [23.8567],
        [28.6816]], grad_fn=<AddBackward0>)
```

```
corrects = (preds>0.0).float() == train_y
corrects
```

```
tensor([[ True],
        [ True],
        [ True],
        ...,
        [False],
        [False],
        [False]])
```

```
corrects.float().mean().item()
```

```
0.4912068545818329
```

- SGD 를 사용하는데 필요한 loss function
  - Prediction 을 할때 사용한 metric 인 accuracy 를 사용했을 때의 문제점
    - Weight 값의 작은 변화를 주어도 전체 prediction 자체는 변하지 않기에 gradient 가 0 이 됨
    - 미세한 weight 의 변화값에 대해 loss function 이 변화해야 미분값을 얻을 수 있음

- 예측값 뿐만 아니라, confidence 도 계산
- 예측값이 실제값과 맞았을 때의 confidence 와 예측값이 실제값과 달랐을 때의 1-confidence 를 이용

```
trgts  = tensor([1,0,1])
prds   = tensor([0.9, 0.4, 0.2])
```

```
torch.where(trgts==1, 1-prds, prds)
```

```
tensor([0.1000, 0.4000, 0.8000])
```

- Confidence level 에 따라 예측값 (pred) 이 같아도 loss function 의 아웃풋이 달라짐

```
mnist_loss(prds,trgts)
```

```
tensor(0.4333)
```

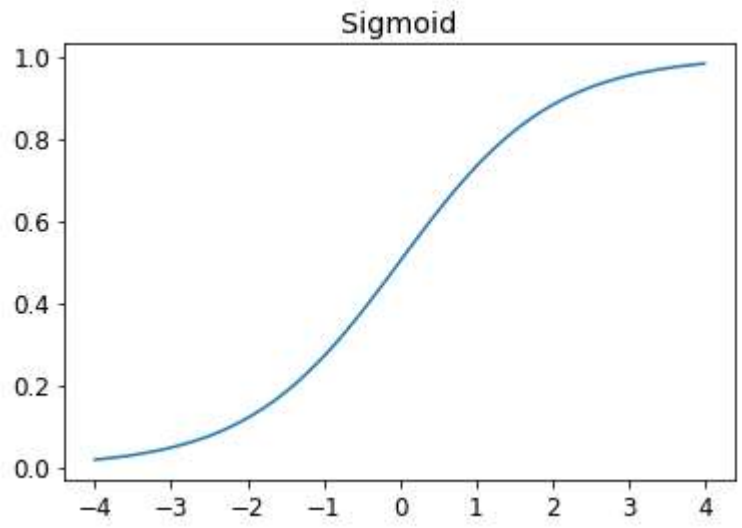
```
mnist_loss(tensor([0.9, 0.4, 0.8]),trgts)
```

```
tensor(0.2333)
```

**Sigmoid**

- Sigmoid 함수는 0-1 사이의 값을 계산함 (아무 input 값을 (양수나 음수) smooth 한 0 ~ 1 값으로 변환)

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




- 함수가 곡선이기 때문에 SGD 에 사용하기 용이함 (예측값이 0 또는 1 처럼 discrete 하지 않음)

```
def mnist_loss(predictions, targets):
    predictions = predictions.sigmoid()
    return torch.where(targets==1, 1-predictions, predictions).mean()
```

- 이 모든 작업 (예측을 confidence 로 표현) 은 학습을 자동화하기에 필요한 수단임
  - 미분이 가능하여야 한다
  - 한 세트에서 loss function 을 evaluate 할 때 각각 epoch 마다 loss 값을 계산하고 epoch 가 끝날 때 그 step 의 loss 평균값을 내어 weight 을 업데이트 한다
  - 이 step 에서 유저가 이해하기 쉬운 metric 을 통해 model performance 도 보여준다 (accuracy)

**SGD and Mini-Batches**

- SGD 를 통해서 학습이 각 step 마다 되었으면, optimization step 을 통해 weight 을 업데이트 한다
  - 전체 학습 데이터를 다 loss 를 계산하고 weight 을 업데이트 하면 efficient 하지 않음
  - 데이터 하나씩 loss 를 evaluate 해도 gradient 가 안정적이지 않거나, 정확하지 않음

- Mini-batch 를 통해 적당히 데이터를 잘라내서 학습시킴 (loss 평균값)
  - 한 epoch 에서 사용되는 데이터의 수
  - 몇개의 데이터를 사용함은 batch size 로 결정
  - 각 epoch 마다 데이터를 shuffle 해서 mini-batch 를 만듬 (DataLoader)

```
coll = range(15)
dl = DataLoader(coll, batch_size=5, shuffle=True)
list(dl)
```

```
[tensor([ 3, 12,  8, 10,  2]),
 tensor([ 9,  4,  7, 14,  5]),
 tensor([ 1, 13,  0,  6, 11])]
```

- 이런 randomize 된 mini-batch 를 형서할 땐 X, Y 가 같이 묶여있는 튜플 형태로 shuffle 해야 함

```
dl = DataLoader(ds, batch_size=6, shuffle=True)
list(dl)
```

```
[(tensor([17, 18, 10, 22,  8, 14]), ('r', 's', 'k', 'w', 'i', 'o')),
 (tensor([20, 15,  9, 13, 21, 12]), ('u', 'p', 'j', 'n', 'v', 'm')),
 (tensor([ 7, 25,  6,  5, 11, 23]), ('h', 'z', 'g', 'f', 'l', 'x')),
 (tensor([ 1,  3,  0, 24, 19, 16]), ('b', 'd', 'a', 'y', 't', 'q')),
 (tensor([2, 4]), ('c', 'e'))]
```