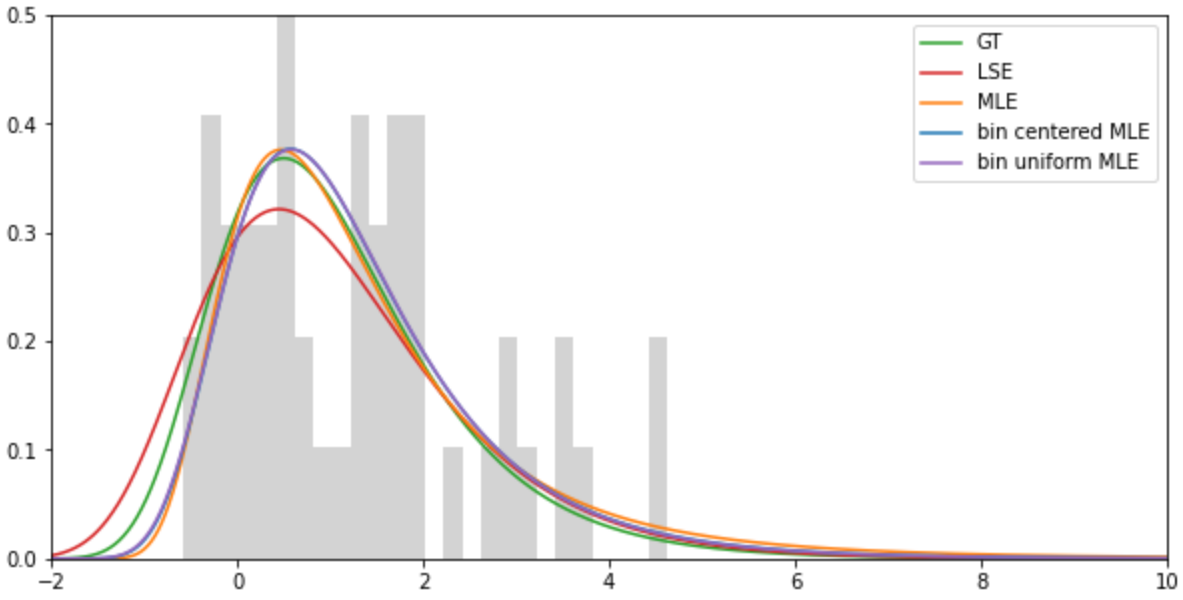
Tips for ML/DL

상태	Basic ML
담당자	

Maximum Likelihood Estimation(MLE)

> Observation을 가장 잘 설명하는 어떤 확률분포함수의 파라미터 θ를 찾아내는 과정

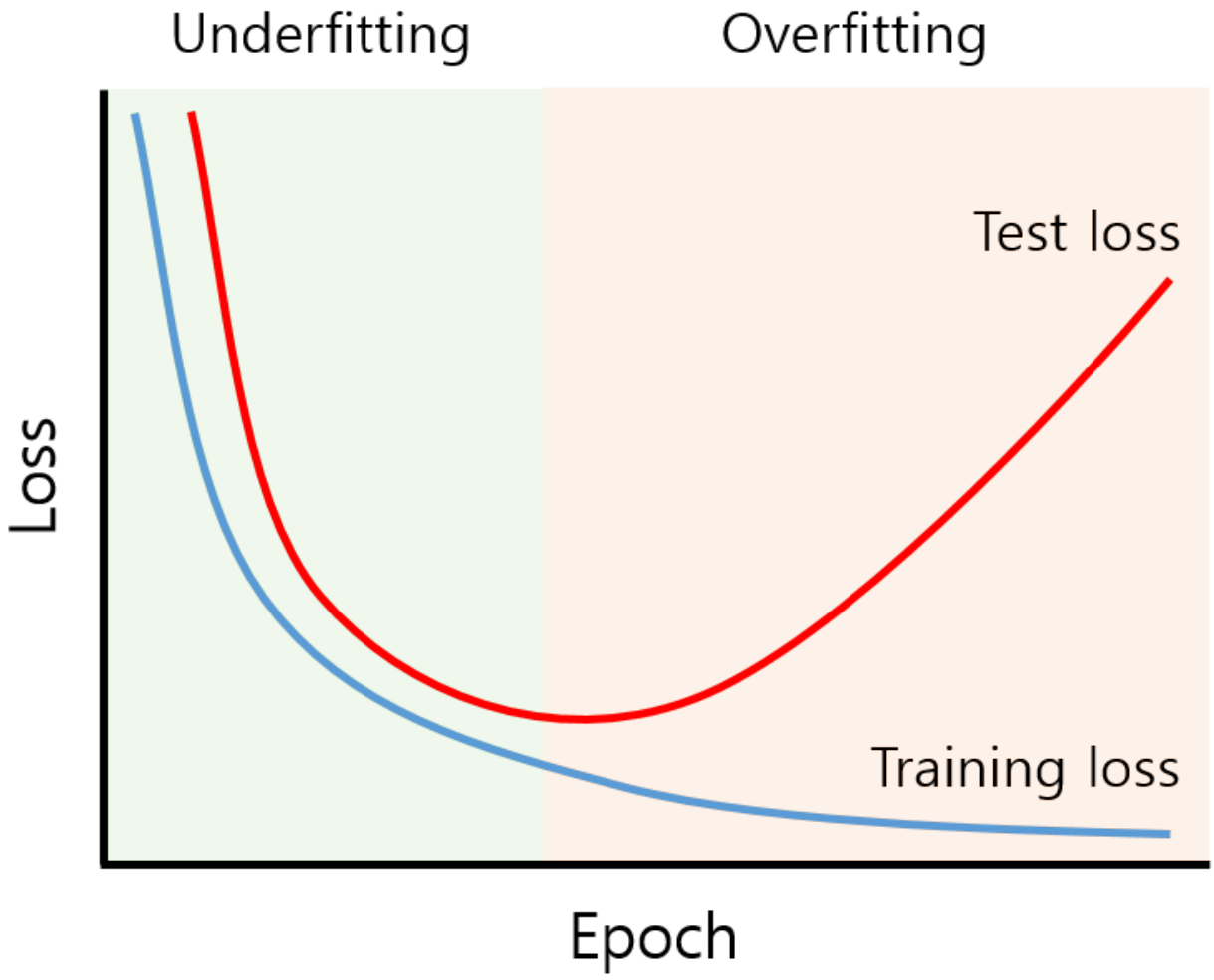


이 파라미터 θ에 대해 Observation이 등장할 확률이 최대화되는지는 어떻게 알 수 있을까? Gradient Ascent 방법을 사용해(Likelihood는 loss와 달리 maximize되어야 하므로) 파라미터의 업데이트 방향을 알 수 있다.

Overfitting

> 주어진 데이터를 가장 잘 설명하는 확률분포를 찾다 보니 발생하는 문제

이를 위해서는 training set, test set을 나누어(보통 training set의 비율이 0.8 정도), training set만 학습시켜 이것이 test set에서도 좋은 성능을 내는지 확인한다.



Train loss는 지속적으로 감소하다 수렴하는 반면 test loss는 감소하다 다시 상승하는데, 이 지점이 바로 overfitting이 시작되는 지점이다. 이상적인 학습 과정은 train과 test 모두에서 loss가 학습에 따라 감소하는 것

- 데이터의 수를 증가시키거나
- Feature의 수를 감소시키거나
- 규제(regularization)를 적용함으로써 (딥러닝에서는 dropout과 batch normalization을 많이 사용)

Overfitting을 방지할 수 있다.

DNN을 학습시키는 전략

1. NN architecture를 설계
2. 학습 후 해당 모델이 오버피팅되었는지 확인
    a. 오버피팅되지 않음 → 모델 사이즈를 키움(더 깊고 넓게)
    b. 오버피팅 됨 → dropout이나 batch normalization 같은 규제를 적용
3. 이를 반복

PyTorch

Overfitting Sample

```
x_train = torch.FloatTensor([[1, 2, 1],
                             [1, 3, 2],
                             [1, 3, 4],
                             [1, 5, 5],
                             [1, 7, 5],
                             [1, 2, 5],
                             [1, 6, 6],
                             [1, 7, 7]]) # 8 x 3 Tensor
y_train = torch.LongTensor([2, 2, 2, 1, 1, 1, 0, 0]) # 8 x 1 Tensor


x_test = torch.FloatTensor([[2, 1, 1],
                            [3, 1, 2],
                            [3, 3, 4]]) # 3 x 3 Tensor
y_test = torch.LongTensor([2, 2, 2]) # 3 x 1 Tensor
```

```
class SoftMaxClassificationModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(3, 3)

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

```
model = SoftMaxClassificationModel()
optimizer = optim.SGD(model.parameters(), lr=0.1)
```

```
def train(model, optimizer, x_train, y_train):
    nb_epochs = 20

    for epoch in range(nb_epochs):
        prediction = model(x_train)
        cost = F.cross_entropy(prediction, y_train)

        optimizer.zero_grad()
        cost.backward()
        optimizer.step()

        print('Epoch : {:4d}/{} Cost : {:.6f}'.format(epoch, nb_epochs, cost.item()))


def test(model, optimizer, x_test, y_test):

    prediction = model(x_test)
    cost = F.cross_entropy(prediction, y_test)
    predicted_classes = prediction.max(1)[1]
    correct_count = (predicted_classes == y_test).sum().item()

    print('Accuracy : {}% Cost : {:.6f}'.format(correct_count / len(y_test) * 100,
                                                cost.item()))
```

```
train(model, optimizer, x_train, y_train)

>>
...
Epoch   19/20 Cost: 0.989365

test(model, optimizer, x_test, y_test)

>>
Accuracy: 0.0% Cost: 1.425844    ## Overfitting
```

Learning Rate

> Learning rate가 너무 크면 발산하면서 cost가 점점 늘어난다. 이를 ovrshooting이라 한다. 반면 learning rate가 너무 작으면 cost가 줄어들지 않는다. ⇒ 적절한 learning rate 선택

▼ `torch.no_grad()`를 사용하기

```
def test(model, optimizer, x_test, y_test):
    with torch.no_grad():
        prediction = model(x_test)
        cost = F.cross_entropy(prediction, y_test)
        predicted_classes = prediction.max(1)[1]
        correct_count = (predicted_classes == y_test).sum().item()

        print('Accuracy : {}% Cost : {:.6f}'.format(correct_count / len(y_test) * 100,
                                                    cost.item()))
```

우리가 gradient를 계산하는 이유는 loss를 minimize하는 방향으로 hypothesis function을 업데이트 하기 위함이다.

테스트 시에는 이 과정이 필요없으므로 `torch.no_grad()`를 명시해 불필요하게 기울기를 계산하는 수고를 덜 수 있다. `time.time()`을 이용해 시간을 비교한 결과 `torch.no_grad()`를 명시한 쪽이 더 빠르게 코드가 실행되었다.

Loss에 `.requires_grad`를 실행하면 True/False로 확인 가능하다.

```
nb_epochs = 20
model = SoftMaxClassificationModel()

start = time.time()
for epoch in range(nb_epochs):
    prediction = model(x_test)
    cost2 = F.cross_entropy(prediction, y_test)
#     print('Epoch {:2d}/{} Cost {:.4f}'.format(epoch, nb_epochs, cost.item()))
print(time.time()-start)

>>
0.004635810852050781

## With torch.no_grad()
nb_epochs = 20
model = SoftMaxClassificationModel()

start = time.time()
for epoch in range(nb_epochs):
    with torch.no_grad():
        prediction = model(x_test)
        cost3 = F.cross_entropy(prediction, y_test)
#         print('Epoch {:2d}/{} Cost {:.4f}'.format(epoch, nb_epochs, cost.item()))
print(time.time()-start)

>>
0.004106283187866211
```

Data Preprocessing

> 데이터를 zero-center하고 normalize하면 성능을 높일 수 있다.

예를 들어 `[[1000, 0.1], [999, 0.2], [1010, 0.3]]` 같은 데이터셋이 주어졌을 때, 데이터를 정규화하여 전처리하면 성능 향상에 도움이 될 수 있다.