20.04.06

주제	**Maximum Likelihood Estimation(MLE)**

이론

Maximum Likelihood Estimation(MLE)

한국 말로는 '최대 가능도 추정'이라고 부른다.

Likelihood란 무엇이고, MLE를 왜 하는 것일까?

압정을 땅에 던졌을 때 압정이 떨어진 모양을 두 가지 케이스로 나누는 걸 생각해보자. 납작한 부분이 바닥에 완전히 닿은 경우 Class 1, 뾰족한 부분이 바닥을 짚는 경우를 Class 2라고 생각해보자. 물론 물리적으로 정확한 확률 분포가 있기야 할 것이다. 하지만 우리는 이걸 머신러닝을 통해 구해보도록 하겠다.

Binomial distribution이기 때문에 압정이 떨어진 모양은 베르누이 분포를 가질 것이다. 이 베르누이 분포를 갖는 동작을 여러 번 반복하면 이항분포로서 나타난다. (아래 수식)

$$K \sim \mathcal{B}(n,\theta) P(K=k) = \binom{n}{k} \theta^k (1-\theta)^{n-k} = \frac{n!}{k!(n-k)!} \cdot \theta^k (1-\theta)^{n-k}$$

n과 k는 우리의 관찰 결과로 채워 넣으면, 이 함수는 theta에 따른 함수인데, 이걸 그래프로 그리면 theta에 따른 likelihood를 나타내는 언덕 모양이 된다. (likelihood 가 확률에 비례하므로)우리는 이 likelihood가 가장 큰 지점의 theta를 찾아야 한다. 그래서 경사하강법과 매우 유사한 방식으로 최대점을 찾는 것이다.(Optimization via Gradient Descent) 대충 이런 모양이 될 거다.

$$\theta := \theta - \alpha \nabla_\theta L(x;\theta)$$

뭐 이런 식으로 θ를 업데이트한다.

만약 연속적이고 가우시안 분포(=정규 분포)였다면 f(θ) 가 아니라 f(μ,σ)로 구해야 할 것이다.

음..... 추후에 보충해야 할 것 같다. 일단 참고할 만한 링크 저장

MLE의 Overfitting 문제

MLE를 쓸 경우, 가장 확률적으로 적합한 모델을 기계적으로 찾는 방법이기 때문에 조금 불균질한, 튀는 데이터에 대해서도 전부 수용할 수 있는 **복잡한 모델**을 내놓을 수가 있다. 이렇게 학습 데이터에 과도하게 딱 맞는 모델이 나오는 상황을 ***Overfitting*** 이라고 부른다.

이를 해결하는 방법에 대해선 밑(<Overfitting 방지>항목)에서 설명하겠다.

Overfitting 실습

모델이 Training set에 대해 과적합되어, 새로운 데이터인 Test Set에는 적용되지 못하는 현상을 관찰해보겠다.

Imports

In [14]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# For reproductibitily
torch.manual_seed(1)
```

Out[14]:

<torch._C.Generator at 0x7f85c45fc750>

Training Set과 Testing Set 나누어서 선언해줌

In [0]:

```
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]])
# |x_train| = (m, 3)

y_train = torch.LongTensor([2, 2, 2, 1, 1, 1, 0, 0])
# |y_train| = (m,  ) One Hot Vector의 인덱스들을 가지고 있음

x_test = torch.FloatTensor([[2, 1, 1], [3, 1, 2], [3, 3, 4]])
# |x_test| = (m', 3)    x_train과 feature 수가 같아야 하므로 똑같이 3

y_test = torch.LongTensor([2, 2, 2])
# |y_test| = (m',  )
```

모델 설정.

이전에 배웠던 Softmax Regression용 모듈을 적용한다.

In [0]:

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

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

model = SoftmaxClassifierModel()

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

관찰하기

Training Set에 대해 위 모듈을 적용해 모델을 학습시키는 함수를 만든다.

```
# Training data에 대해 Seftmax Regression을 20회에 걸쳐 수행하고 결과 출력하는 함수
def train(model, optimizer, x_train, y_train) :
  nb_epochs = 20;
  for epoch in range (nb_epochs):

    # H(x) 계산
    prediction = model(x_train)   # |prediction| = |x_train| = (m, 3)


    # cost 계산
    cost = F.cross_entropy(prediction, y_train)

    # cost로 H(x) 계산

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

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

Test를 통해 Accuracy를 구하는 함수도 만들어준다.

```
def test(model, opt, x_test, y_test):
  prediction = model(x_test)                  # model을 test set에 통과시킴. |x_test| = (m',3)
  predicted_classes = prediction.max(1)[1]    # |prediction| = (m',3)
  # max 함수는 입력된 dimention의 방향에 맞는 최대값을 구해주는 함수였던 것 같다..
  correct_count = (predicted_classes == y_test).sum().item()
  cost = F.cross_entropy(prediction, y_test)  # 정답과의 cross entropy를 구한다.

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

함수를 호출해 작동시키고, 결과를 지켜 본다.

```
train(model, optimizer, x_train, y_train)
test(model, optimizer, x_test, y_test)
```

▼ 실행 결과

E: 0/20 Cost: 2.203667
E: 1/20 Cost: 1.199645
E: 2/20 Cost: 1.142985
E: 3/20 Cost: 1.117769
E: 4/20 Cost: 1.100901
E: 5/20 Cost: 1.089523
E: 6/20 Cost: 1.079872
E: 7/20 Cost: 1.071320
E: 8/20 Cost: 1.063325
E: 9/20 Cost: 1.055720
E: 10/20 Cost: 1.048378
E: 11/20 Cost: 1.041245
E: 12/20 Cost: 1.034285
E: 13/20 Cost: 1.027478
E: 14/20 Cost: 1.020813
E: 15/20 Cost: 1.014279
E: 16/20 Cost: 1.007872
E: 17/20 Cost: 1.001586
E: 18/20 Cost: 0.995419

E: 19/20 Cost: 0.989365
Accuracy: 0.0% Cost: 1.425844

위 결과에서 Trainig Set에 대한 Cost는 점차 낮아지는 것을 관찰 할 수 있으나, Test Set에 대한 Cost는 외려 다소 큰 것을 볼 수 있다. 20 Epoch에 도달했을 지점엔 이미 overfitting이 진행되었다는 걸 알 수 있다.

머신러닝 모델 설계 Tips

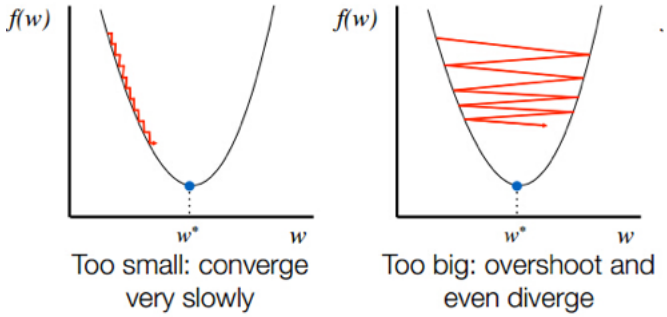
Learning rate 조절

Cost Function을 최소화하기 위한 Gradient Descent Algorithm을 복기해보자.

$$-\alpha \triangle \mathcal{L}(w_1, w_2)$$

에서 α는 프로그램에서 우리가 lr(learning rate)으로 지정해주었던 상수다. 이 α 값을 어느 정도로 정해줘야 가장 괜찮은 모델을 뽑을 수 있을까?

- α가 너무 **작다면** : 학습이 너무 더뎌서 많은 epoch가 필요함
- α가 너무 **크다면** : 최솟값으로 다가가질 않고 건너뛰는, **overshooting** 발생 가능성 있음. (따라서 혹시 cost가 작아지지 않고 계속 발산하기만 한다면 learning rate을 너무 크게 잡진 않았는지 검토해봐야 함.)



학습률 조절에 관한 시각적인 실습을 여기서 간단하게 해볼 수 있다.

검색을 해보니, 차라리 overshooting은 바로 알아차릴 수나 있어서 learning rate이 너무 작은 경우에 비해 더 괜찮은 문제라고 한다. 그리고 적절한 learning rate을 찾는 일반론적인 방법은 딱히 없다고 한다. 여러 번 해보는 게 최선이라고.

Learning Rate 실습

- Gradient Descent에서 learning rate을 너무 **크게** 잡았을 때 overshooting 하는 모습을 관찰해보겠다. lr을 e의 5승(대략 148.4)으로 잡았다.

```
model = SoftmaxClassifierModel()
optimizer = optim.SGD(model.parameters(), lr = 1e5)
train(model, optimizer, x_train, y_train)
```

▼ 실행 결과

E: 0/20 Cost: 1.280268
E: 1/20 Cost: 976950.750000
E: 2/20 Cost: 1279135.000000
E: 3/20 Cost: 1198379.125000
E: 4/20 Cost: 1098825.625000

E: 5/20 Cost: 1968197.625000
E: 6/20 Cost: 284763.125000
E: 7/20 Cost: 1532260.000000
E: 8/20 Cost: 1651504.250000
E: 9/20 Cost: 521878.437500
E: 10/20 Cost: 1397263.125000
E: 11/20 Cost: 750986.250000
E: 12/20 Cost: 918691.750000
E: 13/20 Cost: 1487888.125000
E: 14/20 Cost: 1582260.125000
E: 15/20 Cost: 685818.000000
E: 16/20 Cost: 1140048.750000
E: 17/20 Cost: 940566.750000
E: 18/20 Cost: 931638.125000
E: 19/20 Cost: 1971322.625000

- 이번에는 Learning rate이 너무 작을 때 cost 값이 잘 줄어들지 않는 것을 관찰하겠다.

```
model = SoftmaxClassifierModel()
optimizer = optim.SGD(model.parameters(), lr = 1e-10)
train(model, optimizer, x_train, y_train)
```

▼ 실행 결과

E: 0/20 Cost: 3.187324
E: 1/20 Cost: 3.187324
E: 2/20 Cost: 3.187324
E: 3/20 Cost: 3.187324
E: 4/20 Cost: 3.187324
E: 5/20 Cost: 3.187324
E: 6/20 Cost: 3.187324
E: 7/20 Cost: 3.187324
E: 8/20 Cost: 3.187324
E: 9/20 Cost: 3.187324
E: 10/20 Cost: 3.187324
E: 11/20 Cost: 3.187324
E: 12/20 Cost: 3.187324
E: 13/20 Cost: 3.187324
E: 14/20 Cost: 3.187324
E: 15/20 Cost: 3.187324
E: 16/20 Cost: 3.187324
E: 17/20 Cost: 3.187324
E: 18/20 Cost: 3.187324
E: 19/20 Cost: 3.187324

아무튼 이렇게 학습이 발산을 하거나, 너무 더디게 진행될 경우 lr을 조절해주는 노하우가 필요하다

Data Preprocessing (데이터 선처리)

데이터 선처리를 해야 할 경우가 있다.

- Data preprocessing for gradient descent:
  y = w1*x1* + *w2*x2 + b 와 같은 모델을 이용해 학습하는 경우를 살펴 보자.
  w1, w2를 변수로 갖는 그래프는 대략 한 곳을 최저점으로 삼는 움푹 패인 모양이 될 것임.이런 모양을 갖는 것이 이상적이다.
  그러나 실제로는 x1과 x2의 분산 값이 서로 비슷하지 않다면, 저게 균일한 원형이 되지 않을 것이다.

Fig.1a 3D plot for $J(\theta_1, \theta_2) = (\theta_1^2 + \theta_2^2)$

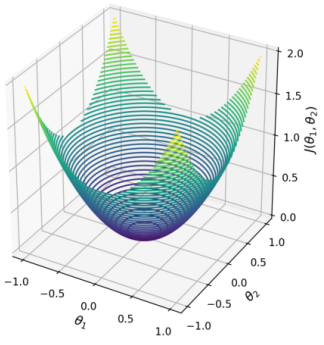


Fig.1b 3D plot - rotated

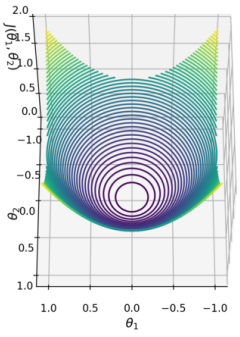
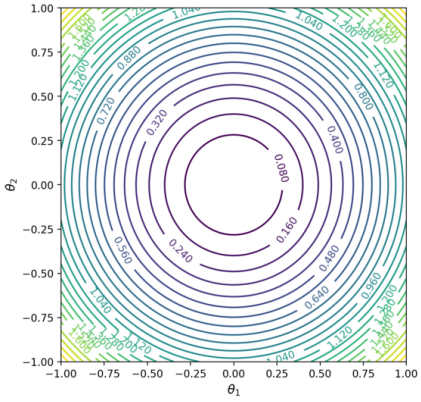
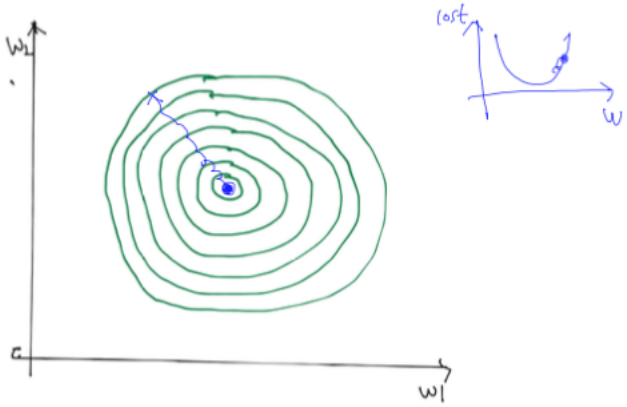


Fig.1c Contour plot for $J(\theta_1, \theta_2) = (\theta_1^2 + \theta_2^2)$

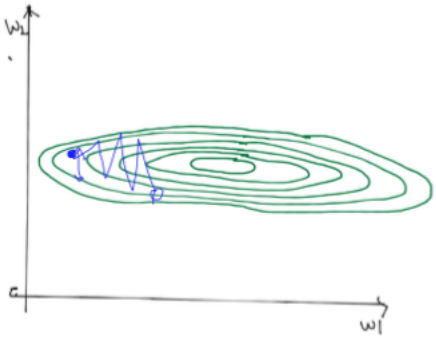


Data (X) preprocessing for gradient descent



Data (X) preprocessing for gradient descent

x1	x2	y
1	9000	A
2	-5000	A
4	-2000	B
6	8000	B
9	9000	C



이런 모양이 될 경우 한 축에 대해서 α 값을 잘 조정해 주었더라도 다른 축에 대해선 너무 큰 값이 되어 위 그림에서 w2축에 대해 overshooting(?)이 발생한 것처럼 cost가 수렴하지 않고 튕겨다니는 걸 볼 수 있다. 이를 해결하기 위해 normalize 를 한다. 다음에 소개하는 것은 그 방법 중 하나다.

- Standardization

  수식으로는,

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

  이렇게 표현되며, 실제 코드로 옮길 땐

```
x_std[:,0] = (x[:,0] - x[:,0].mean) / x[:,0].std()
```

  이런 식이다.

  어디서 많이 봤다 했더니 고등학교 수학에서 정규분포를 표준화하는 공식과 동일한 것 같다.

Data Preprocessing

데이터 전처리(선처리)

In [0]:

```
x_train  = torch.FloatTensor([[73, 80, 75],
                              [93, 88, 93],
                              [89, 91, 90],
                              [96, 98, 100],
                              [73, 66, 70]])
y_train  = torch.FloatTensor([[152],[185],[180], [196], [142]])
```

Mean Square Error 값을 쓰게 될 것임.

아래 코드는 x_train 벡터들을 Normalization 시키는 과정이다.

방법 - Standardization; 정규화

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

In [34]:

```
mu = x_train.mean(dim=0)
sigma = x_train.std(dim=0)
norm_x_train = (x_train - mu)/sigma
print(norm_x_train)

tensor([[-1.0674, -0.3758, -0.8398], [ 0.7418, 0.2778, 0.5863], [ 0.3799, 0.5229, 0.3486], [ 1.0132, 1.0948, 1.1409], [-1.0674, -1.5
```

정규 분포를 따르는 training set을 만들어 주었다.

그 이후엔 데이터에 맞게 regression 모델을 짜주겠다.

```
class MultivariableLinearRegressionModel(nn.Module):
  def __init__(self):
    super().__init__()
    self.linear = nn.Linear(3, 1)
```

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

In [0]:

```
model = MultivariableLinearRegressionModel()
optimizer = optim.SGD(model.parameters(), lr = 1e-1)
```

Training 함수 짜기

In [0]:

```
# Training

def train(model, optimizer, x_train, y_train):
  nb_epochs = 20
  for epoch in range(nb_epochs):

    # H(x) 계산
    prediction = model(x_train)
    # |x_train| = (m, 3)
    # |prediction| = (m, 1)

    # Cost는 MSE로 계산
    cost = F.mse_loss(prediction, y_train)
    # cost를 토대로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

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

In [38]:

```
print("*** Training with Preprocessed Data")
train(model, optimizer, norm_x_train, y_train)
print("*** Without data Preprocessing... ")
model = MultivariableLinearRegressionModel()
optimizer = optim.SGD(model.parameters(), lr = 1e-1)
train(model, optimizer, x_train, y_train)
```

▼ 실행 결과

*** Training with Preprocessed Data
E: 0 / 20 Cost: 29615.740234
E: 1 / 20 Cost: 18803.878906
E: 2 / 20 Cost: 11991.029297
E: 3 / 20 Cost: 7661.868164
E: 4 / 20 Cost: 4900.236816
E: 5 / 20 Cost: 3135.413574
E: 6 / 20 Cost: 2006.681396
E: 7 / 20 Cost: 1284.504639
E: 8 / 20 Cost: 822.366089
E: 9 / 20 Cost: 526.605774
E: 10 / 20 Cost: 337.314728
E: 11 / 20 Cost: 216.160767
E: 12 / 20 Cost: 138.613449
E: 13 / 20 Cost: 88.974701
E: 14 / 20 Cost: 57.197395
E: 15 / 20 Cost: 36.851803

E: 16 / 20 Cost: 23.822887
E: 17 / 20 Cost: 15.476827
E: 18 / 20 Cost: 10.128090
E: 19 / 20 Cost: 6.697937
*** Without data Preprocessing...
E: 0 / 20 Cost: 57621.976562
E: 1 / 20 Cost: 1115690762240.000000
E: 2 / 20 Cost: 21603378185380036608.000000
E: 3 / 20 Cost: 418311281336289389130547200.000000
E: 4 / 20 Cost: 8099858389234773491298680935809024.000000
E: 5 / 20 Cost: inf
E: 6 / 20 Cost: inf
E: 7 / 20 Cost: inf
E: 8 / 20 Cost: inf
E: 9 / 20 Cost: inf
E: 10 / 20 Cost: inf
E: 11 / 20 Cost: inf
E: 12 / 20 Cost: nan
E: 13 / 20 Cost: nan
E: 14 / 20 Cost: nan
E: 15 / 20 Cost: nan
E: 16 / 20 Cost: nan
E: 17 / 20 Cost: nan
E: 18 / 20 Cost: nan
E: 19 / 20 Cost: nan

normalization이 필요한 데이터셋은 어떤 걸까? |y_train|이 (m, 2)와 같이 2차원 prediction 구조를 가질 때, 두 colomn 사이의 수의 스케일이 서로 다르다면 training 모델은 더 큰 값에만 집중을 하게 된다.

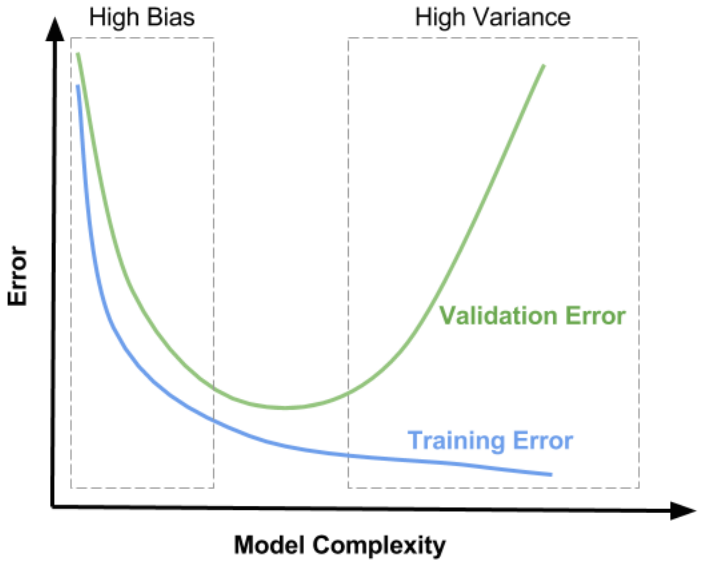
전처리를 수행하게 되면 똑같은 범위의 값으로 바뀔 것이고, 모델은 두가지 데이터에 대해 공정하게 학습을 할 수 있을 것이다.

Overfitting 방지

머신러닝의 가장 큰 골칫거리라고 한다.

'학습 데이터' 그 자체에 너무 딱 맞는 모델로 발전했을 경우를 일컫는다.

학습 데이터로 실험을 해보면 cost도 적고 잘 맞는 것처럼 보이는데, 실전에 투입해서 다른 데이터를 넣어보면 결과가 썩 만족스럽지 않은 현상이 나타난다.



위 그림에서 두 Error(Loss) 간의 거리가 멀어지는 지점을 overfitting 되는 지점으로 생각한다. 성공적인 모델을 만들기 위해선 위 그림에서 High Bias와 High Variance 사이의 구간에 해당하는 모델을 선택해야 한다.

- 해결방법
  - 트레이닝 데이터를 굉~장히 많이 쓴다
    (적은 데이터 셋 환경일 수록 소수의 튀는 값들이 더 큰 영향을 주기 때문. 표본이 많을 수록 모집단과 비슷하다.)
  - feature(데이터를 설명하는 특징)의 개수를 될 수 있는 한 줄여준다.
  - **Regularization(일반화)** - 여러 방법이 있다.
    - *Early stopping* : 모델이 너무 복잡해지기 전에, Validation Loss가 더이상 낮아지지 않는 지점에서 훈련을 중단하는 방법
    - *L2 Regularization* : weight에 너무 큰 수를 넣지 않는 방법. 너무 구불구불하고 자세한 모델을 만들지 않도록...

      Cost 함수를 이렇게 만드는 식이다.

      $$\mathcal{L} = \frac{1}{N}\sum_{i}\mathcal{D}\left(S(Wx_i + b)\right) + \lambda\sum W^2$$

      즉 각각의 element에 대한 W가 클 수록 Cost가 증가하도록 하는 것.

      이 때 λ 를 *regularization strength*라고 부르고, 모델을 단순하게 만드는 걸 얼마나 중요하게 생각하느냐를 반영할 수 있는 상수다.

      - 그렇다면 이 람다 값은 어떻게 정할까?

        위 링크의 Stack Overflow 답변에 의하면, 이 역시 Training data의 일부를 λ=0에서부터 값을 점점 키우면서 돌려 보면서 모델이 예측값을 어떻게 내놓는지를 관찰하며 결정해야 한다. 그리고 결정한 값에서 살짝 작게 잡아야 전체 데이터에 맞을 것...이라고 말하는 것 같다.

        더 나아가면, 람다 값을 직접 관찰하며 임의로 정해 줘야 한다는 모호함을 피하고 싶다면, Tikhonov Regularization 이라는 다른 일반화 방법에선 상수값에 대한 솔루션을 확실하게 정해줄 수 있으니 그 쪽을 권한다는 답변도 있다.
    - 딥러닝에선 Neural Network 크기를 줄이는 방법이 유효하다.
    - 가장 많이 사용되는 Dropout, Batch normalization 이라는 방법이 있는데, 이 역시 딥러닝에서 사용되는 기술이다. 추후에 다루겠다고 함.

DNN에서의 오버피팅 방지

- Deap Neural Network의 과정
  1. 입력 데이터가 1D vector고 10개의 feature가 있다고 하면, 5개 유닛을 가진 softmax layer가 나올 것이다.
  2. 오버피팅이 될 때까지 size를 늘려 나간다. (input, output은 고정한 채로 중간의 깊이와 너비만 확장) : Training Set의 Loss는 낮아지면서, Validation Set의 Loss는 높아지기 시작할 때가 overfitting 될 때이니 이걸 확인하고, regularization 방법(drop-out, batch-normalization)을 추가해준다.
  3. 2의 과정을 반복한다.

머신러닝 모델을 평가하는 방법

학습을 충분히 시킨 모델이 얼마나 적합한지 평가하려면 어떻게 해야 할까?

학습 데이터 세트를 그대로 평가에 갖다 쓰는 건 유효하지 않다.

따라서 모델이 한 번도 접해보지 않은, Test set이라는 새로운 데이터 뭉치가 필요하다.

실전에선 training set과 testing set을 어느 정도 비율로 섞어서 모델을 검증하는 편이라고 한다.

- 정확도(Accuracy) 평가
  - 단순하게도, 예측으로 정답을 얼마나 맞췄는지를 비율로 나타낸다.
  - 이미지 인식 분야의 경우 95~99% 정도의 정확도를 가짐.

Training / Testing Data Set

- Training Set (비율 0.8)
  - Training set : 모델을 학습시킴
  - Validation set (비율 0~0.1) : Development set이라고도 부른다.
    기능 (1) α, λ과 같은 조절하는 역할의 상수들을 모의 시험을 해보며 어떤 값이 제일 좋을지 튜닝함.
    기능 (2) 우리는 궁극적으로는 test set을 잘 맞추는 모델을 설계하길 목표로 하기 때문에, test set과 얼마나 맞는지 따지며 Training하기를 '반복'한다면 Test Set에 대해 과적합이 될 가능성도 있다. 그래서 Training Set에 대해 development set으로 새 데이터에 대한 적용 가능성을 테스트하고, 그 다음에 test set을 접하게 만들면 훨씬 정확한 모델을 만들 수 있다. 하지만 이쪽에 데이터 분량을 얼마나 할당해 줄지는 상황에 따라 다르다!
- Test Set (비율 0.1~0.2)

  학습 단계에선 절대 개입하지 않는, 정확도 평가용 데이터 셋.

Online Learning

방대한 분량의 학습 데이터 셋을 여러 묶음으로 쪼개서 (A, B, C, D, ... 라고 부르겠다.)

A set에 대해 학습을 시키고, 그 모델을 이어받아 B set에 대해 학습을 시키고, 이전 결과를 또 이어받은 상태로 C set도 학습시키고... 하는 방식을 *Online Learning*이라고 부른다.

이전 데이터를 그대로 답습하는 게 아니라 추가로 학습을 시키는 점에서 장점이 있다.

실습

여기다가도 따로 정리하고 싶었는데 시간적/에너지적 여유가 없어서 colab으로 정리한 걸 그대로 가져오도록 하겠음. 과제 링크 건 것과 똑같음.

Lab 07-1 Overfitting & Overshooting

- 내용 :
  1. Training data에 Overfitting되는 걸 확인하는 방법 알아보기
  2. Learning rate을 너무 낮게 / 높게 설정했을 때의 cost값 변화 관찰하기

3. 데이터 전처리 방법을 배우고, 전처리를 하지 않았을 때의 결과와 비교하기

Lab 07-2 How to use MNIST dataset

- 내용 :
  1. MNIST dataset을 import하고 PyTorch에 맞게 가공하는 방법 알아보기
  2. 굉장히 많은 수의 데이터를 불러와서 학습시키고 테스트하는 프로그램의 스크립트 구조 배우기