KU KONKUK UNIVERSITY

Artificial Neural Network

자연어처리연구실

본 챕터의 일부는 김성훈 박사님의 "모두를 위한 머신러닝/딥러닝 강의 (https://hunkim.github.io/ml/)" 자료를 참고하여 편집하였습니다.

PART-II

XOR by PyTorch

실습 코드 다운로드: https://github.com/KUNLP/Lecture

```
import numpy as np
import torch
import torch.nn as nn
from sklearn.metrics import accuracy_score

# 데이터 읽기 함수
def load_dataset(file, device):
  data = np.loadtxt(file)
  print("DATA=",data)

  ?

  input_features = torch.tensor(input_features, dtype=torch.float).to(device)
  labels = torch.tensor(labels, dtype=torch.float).to(device)

  return (input_features, labels)

# 모델 평가 결과 계산을 위해 텐서를 리스트로 변환하는 함수
def tensor2list(input_tensor):
    return input_tensor.cpu().detach().numpy().tolist()
```

train.txt - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

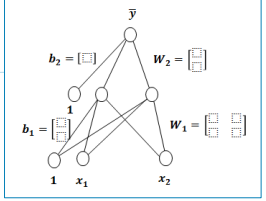
```
0 0 0
0 1 1
1 0 1
1 1 0
```

```
DATA= [[0. 0. 0.]
 [0. 1. 1.]
 [1. 0. 1.]
 [1. 1. 0.]]
INPUT_FEATURES= [[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]]
LABELS= [[0.]
 [1.]
 [1.]
 [0.]]
```


XOR by PyTorch

```
# GPU 사용 가능 여부 확인
if torch.cuda.is_available():
  device = 'cuda'
else:
  device = 'cpu'

input_features, labels = load_dataset("/gdrive/My Drive/colab/ann/xor/train.txt",device)

# NN 모델 만들기
model = nn.Sequential(
          nn.Linear(2, 2, bias=True), nn.Sigmoid(),
          nn.Linear(2, 1, bias=True), nn.Sigmoid()).to(device)

# 이진분류 크로스엔트로피 비용 함수
loss_func = torch.nn.BCELoss().to(device)
# 옵티마이저 함수 (역전파 알고리즘을 수행할 함수)
optimizer = torch.optim.SGD(model.parameters(), lr=1)

# 학습 모드 셋팅
model.train()
```

$$H(X) = \frac{1}{1 + e^{-WTX}}$$



$$Cost(w) = -\frac{1}{m}\sum ylog(H(x)) + (1-y)log(1-H(x))$$

$$W := W - \alpha\frac{\partial}{\partial W}cost(W)$$

XOR by PyTorch

학습 실행 및 코스트 출력

```
# 모델 학습
for epoch in range(1001):

    # 기울기 계산한 것들 초기화
    optimizer.zero_grad()

    # H(X) 계산: forward 연산
    hypothesis = model(input_features)

    # 비용 계산
    cost = loss_func(hypothesis, labels)
    # 역전파 수행
    cost.backward()
    optimizer.step()

    # 100 에폭마다 비용 출력
    if epoch % 100 == 0:
        print(epoch, cost.item())
```

예측(0과 1로 변환) 및 정밀도 계산

```
# 평가 모드 셋팅 (학습 시에 적용했던 드랍 아웃 여부 등을 비적용)
model.eval()

# 역전파를 적용하지 않도록 context manager 설정
with torch.no_grad():
    hypothesis = model(input_features)
    logits = (hypothesis > 0.5).float()
    predicts = tensor2list(logits)
    golds = tensor2list(labels)
    print("PRED=",predicts)
    print("GOLD=",golds)
    print("Accuracy : {0:f}".format(accuracy_score(golds, predicts)))
```

```
0 0.6988072395324707
100 0.693162202835083
200 0.6930413246154785
300 0.692793071269989
400 0.6919294595718384
500 0.6865977048873901
600 0.6376820802688599
700 0.5430010557174683
800 0.3161814212799072
900 0.0973285436630249
1000 0.05161437019705772 4
PRED= [[0.0], [1.0], [1.0], [0.0]]
GOLD= [[0.0], [1.0], [1.0], [0.0]]
Accuracy : 1.000000
```


Wide ANN

Hidden layer를 2*2에서 2*10으로 변경

```
# NN 모델 만들기
?
```

더 빨리 수렴함 (학습 속도는 느려짐)

Widening은 선의 개수를 늘리는 효과!

```
0 0.6988072395324707
100 0.693162202835083
200 0.6930413246154785
300 0.692793071269989
400 0.6919294595718384
500 0.6865977048873901
600 0.6376820802688599
700 0.5430010557174683
800 0.3161814212799072
900 0.0973285436630249
1000 0.05161437019705772 4
PRED= [[0.0], [1.0], [1.0], [0.0]]
GOLD= [[0.0], [1.0], [1.0], [0.0]]
Accuracy : 1.000000
```

```
0 0.7083522081375122
100 0.6920320987701416
200 0.6887232065200806
300 0.6461161971092224
400 0.3004415035247803
500 0.09012892842292786
600 0.04456526041030884
700 0.028334952890872955
800 0.020417045801877975
900 0.015820872038600283
1000 0.012849304825067 52
PRED= [[0.0], [1.0], [1.0], [0.0]]
GOLD= [[0.0], [1.0], [1.0], [0.0]]
Accuracy : 1.000000
```


Shallow ANN

Hidden layer를 없애고 Single-layer Perceptron으로 변경

```
# NN 모델 만들기
model = nn.Sequential(
        nn.Linear(2, 1, bias=True), nn.Sigmoid()).to(device)
```

Non-linear Separable Problem

학습 속도는 빠르지만 10,000 epoch를 수행해도 문제를 풀지 못함

```
0 0.7842277884483337
1000 0.6931471824645996
2000 0.6931471824645996
3000 0.6931471824645996
4000 0.6931471824645996
5000 0.6931471824645996
6000 0.6931471824645996
7000 0.6931471824645996
8000 0.6931471824645996
9000 0.6931471824645996
10000 0.6931471824645996
PRED= [[0.0], [0.0], [0.0], [0.0]]
GOLD= [[0.0], [1.0], [1.0], [0.0]]
Accuracy : 0.500000
```


Deep ANN

Hidden layer 층을 1개에서 2개로 변경

```
# NN 모델 만들기
?
```

오래 걸리지만 학습됨

Deeping은 선을 구부리는 효과!

```
0 0.695355892181396 5
100 0.6930767893791199
200 0.6930528283119202
300 0.693021297454834
400 0.6929775476455688
500 0.6929132342338562
600 0.6928118467330933
700 0.692637026309967
800 0.6922696918115234
900 0.6915085315704346
1000 0.6891056299209595
PRED= [[1.0], [0.0], [1.0], [0.0]]
GOLD= [[0.0], [1.0], [1.0], [0.0]]
Accuracy : 0.500000
```

```
0 0.7725627422332764
300 0.6930560469627 38
600 0.6929765939712524
900 0.692727804183 96
1200 0.691035151481628 4
1500 0.580551445484161 4
1800 0.085045225918293
2100 0.014176958240568638
2400 0.007633837871253490 4
2700 0.005209837574511766 4
3000 0.003949393518269062
PRED= [[0.0], [1.0], [1.0], [0.0]]
GOLD= [[0.0], [1.0], [1.0], [0.0]]
Accuracy : 1.000000
```

Deeper ANN

Hidden layer 층을 1개에서 7개로 변경

```
# NN 모델 만들기
model = nn.Sequential(
          nn.Linear(2, 10, bias=True), nn.Sigmoid(),
          nn.Linear(10, 10, bias=True), nn.Sigmoid(),
          nn.Linear(10, 10, bias=True), nn.Sigmoid(),
          nn.Linear(10, 10, bias=True), nn.Sigmoid(),
          nn.Linear(10, 10, bias=True), nn.Sigmoid(),
          nn.Linear(10, 10, bias=True), nn.Sigmoid(),
          nn.Linear(10, 10, bias=True), nn.Sigmoid(),
          nn.Linear(10, 1, bias=True), nn.Sigmoid()).to(device)
```

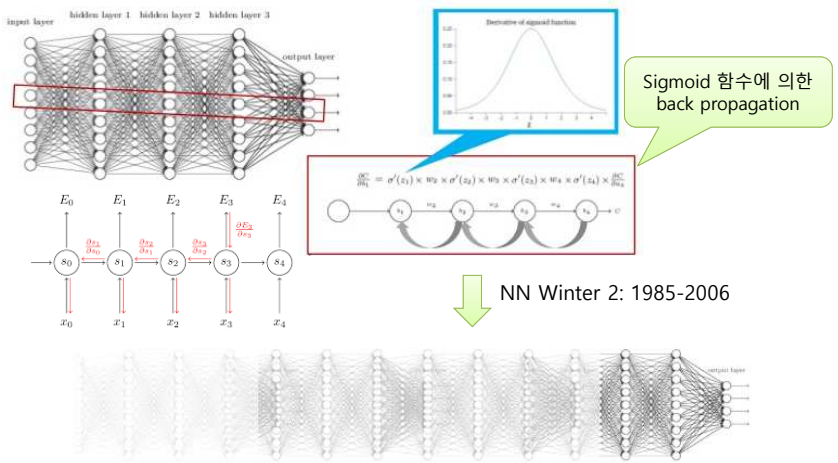
WHY?

10,000 epoch를 돌려도 학습이 안됨!

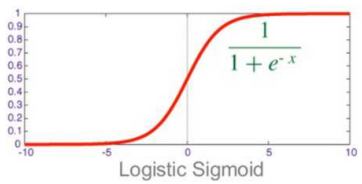
```
0 0.7086373567581177
100 0.6931471824645996
200 0.6931471824645996
300 0.6931471824645996
400 0.6931471824645996
500 0.6931471824645996
600 0.6931471824645996
700 0.6931471824645996
800 0.6931471824645996
900 0.6931471824645996
1000 0.6931472420692444
PRED= [[1.0], [0.0], [0.0], [1.0]]
GOLD= [[0.0], [1.0], [1.0], [0.0]]
Accuracy : 0.000000
```

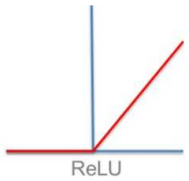
```
0 0.6933478713035583
1000 0.6931472420692444
2000 0.6931472420692444
3000 0.6931471824645996
4000 0.6931471824645996
5000 0.6931471824645996
6000 0.6931471824645996
7000 0.6931471824645996
8000 0.6931471824645996
9000 0.6931471824645996
10000 0.6931471824645996
PRED= [[0.0], [1.0], [0.0], [0.0]]
GOLD= [[0.0], [1.0], [1.0], [0.0]]
Accuracy : 0.750000
```


Vanishing Gradient



Sigmoid to ReLU





Rectified Linear Unit

```
# NN 모델 만들기
?
```

Sigmoid를 ReLU로 변경

마지막 layer는 0과 1사이 값을 출력하도록 하기 위해서 Sigmoid 유지

Sigmoid to ReLU

Vanishing gradient 문제가 사라짐

Sigmoid (10,000 epoch)

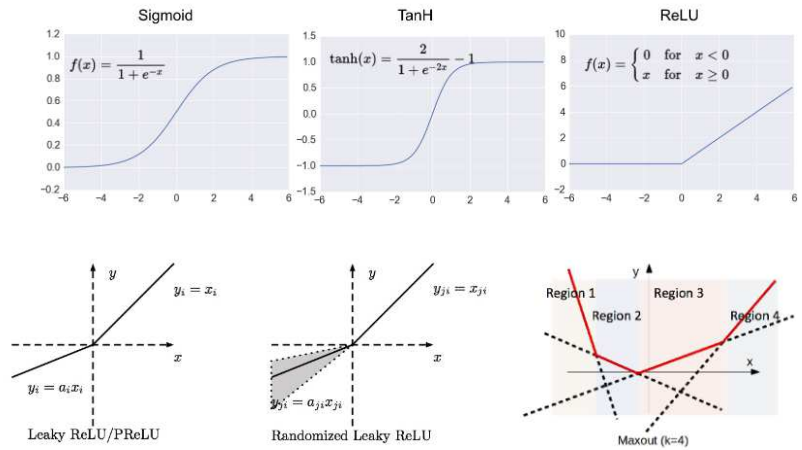
```
0 0.6933478713035583
1000 0.6931472420692444
2000 0.6931472420692444
3000 0.6931471824645996
4000 0.6931471824645996
5000 0.6931471824645996
6000 0.6931471824645996
7000 0.6931471824645996
8000 0.6931471824645996
9000 0.6931471824645996
10000 0.6931471824645996
PRED= [[0.0], [1.0], [0.0], [0.0]]
GOLD= [[0.0], [1.0], [1.0], [0.0]]
Accuracy : 0.750000
```

VS.

ReLU (3,000 epoch)

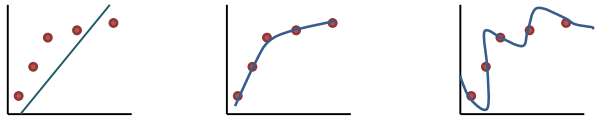
```
0 0.6946406960487366
300 0.6931304931640625
600 0.6931031942367554
900 0.6930624842643738
1200 0.6929516196250916
1500 0.6924918293952942
1800 0.6677674055099487
2100 0.0012112632393836975
2400 0.0003007405321113765
2700 0.0001493549498263746 5
3000 9.424101881450042e-05
PRED= [[0.0], [1.0], [1.0], [0.0]]
GOLD= [[0.0], [1.0], [1.0], [0.0]]
Accuracy : 1.000000
```

Activation Functions



Fitting

- 적합



Underfitting
(high bias)

More training!

Overfitting
(high variance)

More training data
Reduce the number of features
Regularization

Regularization

- Regularization (정규화, 일반화)
  - Cost function 값이 작아지는 방향으로 학습하는 과정에서 특정 가중치가 너무 커져서 일반화 성능이 떨어지는 것을 방지하기 위한 방법
  - L1 regularization

$$D(S,L) = -\frac{1}{N}\sum_i L_i log(S_i) + \lambda \sum |W|$$

Feature selection → Sparse Model에 적합

$a = [0.1, 0.5, 0.2]$ → $||a|| = 8$
$b = [0.3, 0.5, 0.0]$ → $||b|| = 8$

  - L2 Regularization

$$D(S,L) = -\frac{1}{N}\sum_i L_i log(S_i) + \lambda \sum W^2$$

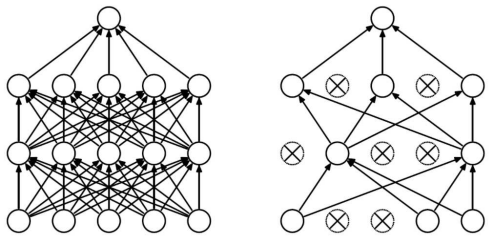
Weight decay

$a = [0.1, 0.5, 0.2]$ → $||a||_2 = 0.55$
$b = [0.3, 0.5, 0.0]$ → $||b||_2 = 0.58$

  - Early Stopping
    - Dev set에서 성능이 더 이상 증가하지 않을 때 지정 횟수보다 학습을 일찍 끝마치는 것
  - Dropout

Dropout

- 드롭아웃
  - 학습 과정 중에 지정된 비율로 임의의 연결을 끊음으로써 일반화 성능을 개선하는 방법



(a) Standard Neural Net (b) After applying dropout.

그림 출처: https://medium.com/@gopalkalpande/biological-inspiration-of-convolutional-neural-network-cnn-9419668898ac

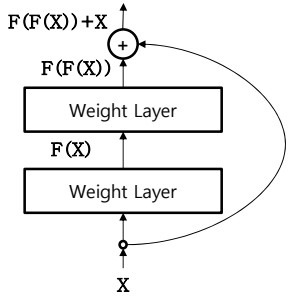
Dropout by PyTorch

```
# NN 모델 만들기
model = nn.Sequential(
          nn.Linear(2, 10, bias=True), nn.ReLU(), nn.Dropout(0.1),
          nn.Linear(10, 10, bias=True), nn.ReLU(), nn.Dropout(0.1),
          nn.Linear(10, 10, bias=True), nn.ReLU(), nn.Dropout(0.1),
          nn.Linear(10, 10, bias=True), nn.ReLU(), nn.Dropout(0.1),
          nn.Linear(10, 10, bias=True), nn.ReLU(), nn.Dropout(0.1),
          nn.Linear(10, 10, bias=True), nn.ReLU(), nn.Dropout(0.1),
          nn.Linear(10, 1, bias=True), nn.Sigmoid()).to(device)

# 이진분류 크로스엔트로피 비용 함수
loss_func = torch.nn.BCELoss().to(device)
# 옵티마이저 함수 (역전파 알고리즘을 수행할 함수)
optimizer = torch.optim.SGD(model.parameters(), lr=0.2)

# 학습 모드 셋팅
model.train()

# 평가 모드 셋팅 (학습 시에 적용했던 드랍 아웃 여부 등을 비적용)
model.eval()

# 역전파를 적용하지 않도록 context manager 설정
with torch.no_grad():
    hypothesis = model(input_features)
```

```
DATA= [[0. 0. 0.]
 [0. 1. 1.]
 [1. 0. 1.]
 [1. 1. 0.]]
INPUT_FEATURES= [[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]]
LABELS= [[0.]
 [1.]
 [1.]
 [0.]]
0 0.7056415677070618
300 0.6939449310302734
600 0.6798115968704224
900 0.6760240197181702
1200 0.7288740873336792
1500 0.5377387404441833
1800 0.4900955557823181
2100 0.004302245099099749
2400 0.2083825021982193
2700 0.6571221351623535
3000 0.0671871230006218
PRED= [[0.0], [1.0], [1.0], [0.0]]
GOLD= [[0.0], [1.0], [1.0], [0.0]]
Accuracy : 1.000000
```


Residual Connection

- 추가 연결
  - 가중치층을 우회하여 상위 층으로 직접 연결하는 것
  - 추상화 정도(낮은 수준 추상화와 높은 수준 추상화)를 적절히 섞어주는 효과 → 앙상블 효과를 통해 성능 개선



질의응답

Q&A

Homepage: http://nlp.konkuk.ac.kr
E-mail: nlpdrkim@konkuk.ac.kr