다층 퍼셉트론

다층 퍼셉트론 신경망 구조

다층 퍼셉트론 신경망은 복수의 퍼셉트론 계층을 순서를 두고 배치하여 입력 벡터로부터 **은닉 계층(Hidden Layer)**를 거쳐 출력 벡터를 얻어내는 신경망 구조이다.

- 같은 계층(Layer) 안에 속한 퍼셉트론들은 동일한 입력을 공유하면서 각각 출력 성분을 만들어내지만 서로 연결이 없어 영향을 주고받을 수 없다.
- 이와 반대로, 인접한 계층(Layer)끼리는 앞 계층의 출력이 뒤 계층의 모든 퍼셉트론에 공통 입력으로 제공된다. (완전 연결 방식)

다층 퍼셉트론 신경망은 단층 퍼셉트론 구조에 비해 **기억 용량**이나, **계산 능력**에 대한 부담이 커지는 대신 **성능 향상**을 기대할 수 있다. (항상 성능이 향상되는 것은 아니다.)

은닉 계층의 수와 폭

출력 계층은 원래의 목표에 따라 고정된 형태를 지니는 경우가 많은데, 은닉 계층(Hidden Layer)에는 이런 제약이 없다. 출력 계층과 달리 은닉 계층의 수와 각 은닉 계층의 폭은 자유롭게 정할 수 있다. 퍼셉트론은 노드라고도 한다.

입력 퍼셉트론 M 개에 연결된 이어지는 Hidden Layer 의 퍼셉트론 N 개를 갖는 한 계층의 파라미터 수는

M * N + N (M * N : 가중치 파라미터, N : 편향 파라미터)

개가 된다.

무조건 단층 퍼셉트론보다 다층 퍼셉트론이 성능이 좋은 것은 아니며, 이와 마찬가지로 무조건 은닉 계층의 수와 폭을 늘린다고 성능이 좋아지는 것은 아니다. **충분한 양**의 **양질의 데이터**가 뒷받침되지 않는다면, 다층 퍼셉트론 구조의 무분별한 확장은 오히려 성능을 떨어뜨릴 수 있다.

따라서 은닉 계층의 수와 폭은 문제의 규모, 데이터 양, 난이도를 종합적으로 고려하여 결정해야 한다. 또한 은닉 계층의 수와 폭을 쉽게 바꿀 수 있도록 하여 최적 값을 빠르게 도출할 수 있게 구현해야 한다.

비선형 활성화 함수

은닉 계층은 가중치와 편향을 이용해 계산된 선형 연산 결과를 바로 출력으로 내보내는 대신 한 번 더 변형시켜 내보낸다. 선형 연산 결과 뒤에 적용되어 퍼셉트론의 출력을 변형시키려고 추가한 장치를 **비선형 활성화 함수**라고 한다.

선형 연산 결과는 곱셈과 덧셈으로만 이루어지는 선형 연산의 특성 상 항상 입력의 **일차 함수**로 나타나는데, 비선형 함수는 **일차 함수로 표현이 불가능한 좀 더 복잡한 기능을 수행하는 함수**이다.

이전에 sigmoid 함수나 softmax 함수 적용 시 이들을 퍼셉트론 내의 구성요소로 간주하지 않았고, 출력 이후에 비선형 함수를 이용하는 방식으로 사용하였는데, 이는 출력 계층의 경우 출력 유형에 따라 후처리 과정에서 비선형 함수를 이용하는 방법이 달라지고, 이 때문에 비 선형 함수를 퍼셉트론 안에 두기가 곤란하기 때문이다. 대표적으로 softmax 함수는 출력 여러 개를 묶어 함께 처리해야 하며 따라서 각기 독립된 구조인 개별 퍼셉트론 안에 이 함수를 나누어 넣을 수가 없다.

결론적으로 출력 계층에는 비선형 활성화 함수가 사용되지 않는다. 그래서 앞서 실습한 단층 퍼셉트론의 예제들의 경우에는 후처리 과정에서 sigmoid 함수나 softmax 함수를 이용했을 뿐, 출력 계층 내부에는 비선형 활성화 함수를 사용하지 않았다.

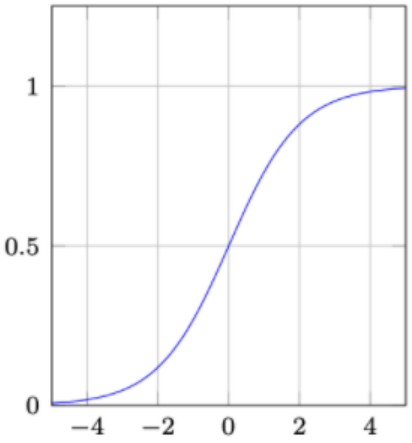
다층 퍼셉트론에서 비선형 활성화 함수는 필수적 구성 요소이다. 다층 퍼셉트론을 도입해도 비선형 활성화 함수를 사용하지 않으면 아무런 의미가 없다. **선형 처리는 아무리 여러 단계를 반복해도 하나의 선형처리**로 줄여서 표현 가능하다는 수학적 원리로 인해 **비선형 활성화 함수가 없는 다층 퍼셉트론은 여전히 선형**이므로 아무런 의미가 없다.

앞서 실습해보았던 민스키의 XOR 문제 역시, 민스키가 XOR 함수를 구현할 수 없다고 한 이유는 선형 처리를 아무리 많이 수행한다고 해도 비선형 처리가 되는 것이 아니라 결국 선형 처리일 뿐이므로 제자리 걸음이라고 생각했기 때문인데, 이를 우리는 다층 퍼셉트론과 비선형 활성화 함수를 적용시켜서 구현할 수 있었다.

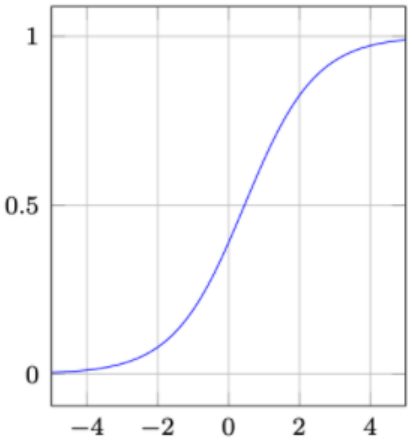
적절한 비선형 함수를 도입하여 이런 선형성의 한계를 극복할 수 있다. 비선형 활성화 함수를 갖춘 은닉 계층을 충분한 수의 퍼셉트론으로 구성하면 단 두 계층의 다층 퍼셉트론 구조만으로 어떤 수학적 함수든 원하는 오차 수준 이내로 동작하게 만들 수 있다. -> 증명됨.

단층 퍼셉트론 구조에 많은 노드 수를 적용하는 것보다 적은 노드 수와 많은 층으로 구성된 다층 퍼셉트론 구조가 더 효과적이다. 이런 점에서 딥러닝이 부각되었다.

ReLu 함수



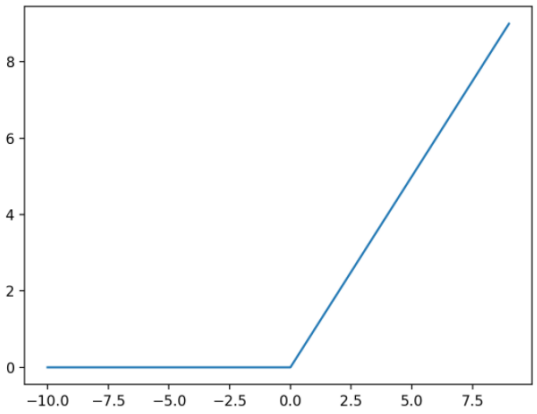
[그림 1] Sigmoid 활성화 함수



[그림 2] Softmax 활성화 함수

sigmoid 와 softmax 도 비선형 함수이지만, 지수 연산이 포함된 복잡한 계산과정 때문에 처리 부담이 크다. 반면 ReLu 함수는 쉽고 빠르게 연산이 가능하다.

ReLu 함수의 한가지 문제는 **x = 0 에서 미분이 불가**능하다는 점인데, 이를 **x = 0 에서 미분값을 강제로 정해줘야 한다**. 보통 0 으로 정해주는 경우가 많다.



[그림 3] ReLu 활성화 함수 그래프

코드

```
def init_model_hidden1():
    global pm_output, pm_hidden, input_cnt, output_cnt, hidden_cnt

    pm_hidden = alloc_param_pair([input_cnt, hidden_cnt]) # 각각 입력과 출력
    pm_output = alloc_param_pair([hidden_cnt, output_cnt]) # 각각 입력과 출력

def alloc_param_pair(shape):
# 무작위 표본 추출. parameter : 평균, 표준편차, size
    weight = np.random.normal(RND_MEAN, RND_STD, shape)
    bias = np.zeros(shape[-1]) # 마지막 차원을 나타냄
    return {'w':weight, 'b':bias} # 가중치와 bias 를 리턴

def forward_neuralnet_hidden1(x):
    global pm_output, pm_hidden # init_model_hidden1 에서 선언한 전역변수 이용
    # 은닉 계층 출력
    hidden = relu(np.matmul(x, pm_hidden['w']) + pm_hidden['b'])
    # 최종 출력
    output = np.matmul(hidden, pm_output['w']) + pm_output['b']

    return output, [x, hidden]

def relu(x): # 은닉 계층 처리 시 비선형 활성화 함수 적용
    return np.maximum(x, 0)

def backprop_neuralnet_hidden1(G_output, aux): # 앞서 배웠던 손실함수 계산에 Hidden
Layer 를 추가한 것.
    global pm_output, pm_hidden

    x, hidden = aux

    g_output_w_out = hidden.transpose()
    G_w_out = np.matmul(g_output_w_out, G_output)
    G_b_out = np.sum(G_output, axis=0)

    g_output_hidden = pm_output['w'].transpose()
    G_hidden = np.matmul(G_output, g_output_hidden)

    pm_output['w'] -= LEARNING_RATE * G_w_out
    pm_output['b'] -= LEARNING_RATE * G_b_out

    G_hidden = G_hidden * relu_derv(hidden)

    g_hidden_w_hid = x.transpose()
```

```
    G_w_hid = np.matmul(g_hidden_w_hid, G_hidden)
    G_b_hid = np.sum(G_hidden, axis=0)

    pm_hidden['w'] -= LEARNING_RATE * G_w_hid
    pm_hidden['b'] -= LEARNING_RATE * G_b_hid

def relu_derv(y):
    return np.sign(y) # x <= 0 이면 0, x > 0 이면 x

# 쉽게 Hidden Layer 의 수와 폭을 조정하기 위한 함수
def init_model_hiddens():
    global pm_output, pm_hiddens, input_cnt, output_cnt, hidden_config

    pm_hiddens = []
    prev_cnt = input_cnt # input -> 첫 Hidden Layer

    for hidden_cnt in hidden_config:
        pm_hiddens.append(alloc_param_pair([prev_cnt, hidden_cnt]))
        prev_cnt = hidden_cnt

    pm_output = alloc_param_pair([prev_cnt, output_cnt]) # 마지막 Hidden Layer ->
output

def forward_neuralnet_hiddens(x):
    global pm_output, pm_hiddens

    hidden = x # 처음 input 값으로 초기화
    hiddens = [x]

    for pm_hidden in pm_hiddens:
       # 입력이자 출력인 값으로 이용
        hidden = relu(np.matmul(hidden, pm_hidden['w']) + pm_hidden['b'])
        hiddens.append(hidden)
    # 출력
    output = np.matmul(hidden, pm_output['w']) + pm_output['b']

    return output, hiddens

def backprop_neuralnet_hiddens(G_output, aux):
    global pm_output, pm_hiddens

    hiddens = aux # 가독성을 위해 hiddens 로 조정

    g_output_w_out = hiddens[-1].transpose()
    G_w_out = np.matmul(g_output_w_out, G_output)
    G_b_out = np.sum(G_output, axis=0)
```

```
    g_output_hidden = pm_output['w'].transpose()
    G_hidden = np.matmul(G_output, g_output_hidden)

    pm_output['w'] -= LEARNING_RATE * G_w_out
    pm_output['b'] -= LEARNING_RATE * G_b_out

    # forward 와는 반복문 순서 거꾸로 되도록 python 의 reversed() 함수를 이용하여 조정
    for n in reversed(range(len(pm_hiddens))):
        G_hidden = G_hidden * relu_derv(hiddens[n+1])

        g_hidden_w_hid = hiddens[n].transpose()
        G_w_hid = np.matmul(g_hidden_w_hid, G_hidden)
        G_b_hid = np.sum(G_hidden, axis=0)

        g_hidden_hidden = pm_hiddens[n]['w'].transpose()
        G_hidden = np.matmul(G_hidden, g_hidden_hidden)

        pm_hiddens[n]['w'] -= LEARNING_RATE * G_w_hid
        pm_hiddens[n]['b'] -= LEARNING_RATE * G_b_hid

global hidden_config # 이 값의 설정여부에 따라 단순 다층 혹은 가변적 다층 퍼셉트론을 선택적으로 호출

def init_model():
    if hidden_config is not None:
        print('은닉 계층 {}개를 갖는 다층 퍼셉트론이 작동되었습니다.'. \
              format(len(hidden_config)))
        init_model_hiddens()
    else:
        print('은닉 계층 하나를 갖는 다층 퍼셉트론이 작동되었습니다.')
        init_model_hidden1()

def forward_neuralnet(x):
    if hidden_config is not None:
        return forward_neuralnet_hiddens(x)
    else:
        return forward_neuralnet_hidden1(x)

def backprop_neuralnet(G_output, hiddens):
    if hidden_config is not None:
        backprop_neuralnet_hiddens(G_output, hiddens)
    else:
        backprop_neuralnet_hidden1(G_output, hiddens)

def set_hidden(info): # 은닉 계층 구조 지정
    global hidden_cnt, hidden_config
    if isinstance(info, int):
```

```
        hidden_cnt = info
        hidden_config = None
    else:
        hidden_config = info
```

1. 전복 고리 수 추정 문제

```
%run ../chap01/abalone.ipynb
%run ../chap04/mlp.ipynb
set_hidden([])
abalone_exec()
```

```
은닉 계층 0개를 갖는 다층 퍼셉트론이 작동되었습니다.
Epoch 1: loss=33.875, accuracy=0.557/0.812
Epoch 2: loss=8.226, accuracy=0.820/0.814
Epoch 3: loss=7.582, accuracy=0.812/0.809
Epoch 4: loss=7.475, accuracy=0.808/0.811
Epoch 5: loss=7.395, accuracy=0.810/0.809
Epoch 6: loss=7.328, accuracy=0.808/0.810
Epoch 7: loss=7.269, accuracy=0.808/0.811
Epoch 8: loss=7.217, accuracy=0.808/0.812
Epoch 9: loss=7.175, accuracy=0.810/0.810
Epoch 10: loss=7.135, accuracy=0.809/0.810

Final Test: final accuracy = 0.810
```

- 은닉 계층 0 개인 단층 퍼셉트론을 기준으로 하여 다층 퍼셉트론의 적용 결과를 확인해보았다.

```
set_hidden(4)
LEARNING_RATE = 0.001
abalone_exec(epoch_count=50, report=10)
```

```
은닉 계층 하나를 갖는 다층 퍼셉트론이 작동되었습니다.
Epoch 10: loss=6.662, accuracy=0.809/0.803
Epoch 20: loss=6.144, accuracy=0.817/0.812
Epoch 30: loss=5.512, accuracy=0.827/0.821
Epoch 40: loss=5.058, accuracy=0.835/0.841
Epoch 50: loss=4.902, accuracy=0.839/0.843

Final Test: final accuracy = 0.843
```

- 은닉 계층 1 개로 4 개의 퍼셉트론을 이용하고 학습률을 0.841 의 정확도를 얻을 수 있었다.
- 단층 퍼셉트론만으로도 비슷한 결과를 얻을 수 있어서 눈에 띄는 정확도 향상은 확인할 수 없었다.

```
set_hidden([3, 3])
abalone_exec(epoch_count=50, report=10)
```

```
은닉 계층 2개를 갖는 다층 퍼셉트론이 작동되었습니다.
Epoch 10: loss=10.554, accuracy=0.732/0.733
Epoch 20: loss=10.553, accuracy=0.730/0.733
Epoch 30: loss=10.554, accuracy=0.731/0.733
Epoch 40: loss=10.554, accuracy=0.731/0.733
Epoch 50: loss=10.553, accuracy=0.731/0.733

Final Test: final accuracy = 0.733
```

- 은닉 계층을 2 개로 늘여 다층 퍼셉트론을 적용한 결과 정확성이 오히려 감소한 것을 확인할 수 있었다. 데이터의 양이 충분하지 않을 시에는 다층 퍼셉트론을 적용시켰을 때 오히려 품질이 감소한다는 것을 눈으로 확인할 수 있었다.

2. 천체 펄서 판정 문제

- 데이터의 양이 편향되어 있어 정확성만으로는 판단할 수 없기 때문에 Accuracy, Precision, Recall, F1 Score 순으로 확인할 수 있도록 출력하였다.

```
pulsar_exec()
```

```
Epoch 1: loss=0.139, result=0.965,0.975,0.624,0.761
Epoch 2: loss=0.128, result=0.971,0.969,0.696,0.810
Epoch 3: loss=0.130, result=0.973,0.951,0.734,0.828
Epoch 4: loss=0.131, result=0.973,0.917,0.762,0.832
Epoch 5: loss=0.130, result=0.975,0.953,0.755,0.843
Epoch 6: loss=0.127, result=0.974,0.893,0.809,0.849
Epoch 7: loss=0.130, result=0.972,0.950,0.718,0.818
Epoch 8: loss=0.119, result=0.974,0.960,0.743,0.837
Epoch 9: loss=0.116, result=0.974,0.971,0.727,0.832
Epoch 10: loss=0.120, result=0.967,0.976,0.649,0.780

Final Test: final result = 0.967,0.976,0.649,0.780
```

- 은닉 계층 0 개인 단층 퍼셉트론을 기준으로 하여 다층 퍼셉트론의 적용 결과를 확인해보았다.

```
%run ../chap02/pulsar_ext.ipynb
%run ../chap04/mlp.ipynb
set_hidden(6)
pulsar_exec(epoch_count=50, report=10)
```

```
은닉 계층 하나를 갖는 다층 퍼셉트론이 작동되었습니다.
Epoch 10: loss=0.093, result=0.972,0.943,0.724,0.819
Epoch 20: loss=0.089, result=0.972,0.955,0.724,0.824
Epoch 30: loss=0.088, result=0.974,0.925,0.774,0.843
Epoch 40: loss=0.086, result=0.975,0.926,0.787,0.851
Epoch 50: loss=0.086, result=0.974,0.959,0.737,0.833

Final Test: final result = 0.974,0.959,0.737,0.833
```

- 은닉 계층 하나를 갖는 다층 퍼셉트론에서 정확성에서는 큰 차이 없는 것으로 보이나, F1 Score 를 보면 확연하게 차이가 있음을 알 수 있다.

```
set_hidden([12,6])
pulsar_exec(epoch_count=200, report=40)
```

```
은닉 계층 2개를 갖는 다층 퍼셉트론이 작동되었습니다.
Epoch 40: loss=0.092, result=0.972,0.929,0.778,0.847
Epoch 80: loss=0.089, result=0.973,0.938,0.778,0.851
Epoch 120: loss=0.087, result=0.974,0.971,0.761,0.854
Epoch 160: loss=0.086, result=0.975,0.949,0.793,0.864
Epoch 200: loss=0.086, result=0.975,0.928,0.804,0.861

Final Test: final result = 0.975,0.928,0.804,0.861
```

- 은닉 계층 2 개를 갖는 다층 퍼셉트론에서는 이전의 은닉 계층 하나를 갖는 다층 퍼셉트론과 정확성 면에서는 큰 차이를 볼 수 없으나 F1 score 에서는 확연히 좋은 성능을 보이는 것을 확인할 수 있다.

3. 철판의 불량 판별 문제

```
steel_exec()
```

```
Epoch 1: loss=15.984, accuracy=0.306/0.320
Epoch 2: loss=15.509, accuracy=0.326/0.197
Epoch 3: loss=15.984, accuracy=0.306/0.348
Epoch 4: loss=15.004, accuracy=0.348/0.197
Epoch 5: loss=15.286, accuracy=0.336/0.202
Epoch 6: loss=15.390, accuracy=0.332/0.440
Epoch 7: loss=15.509, accuracy=0.326/0.442
Epoch 8: loss=15.628, accuracy=0.321/0.455
Epoch 9: loss=15.360, accuracy=0.333/0.322
Epoch 10: loss=15.316, accuracy=0.335/0.455

Final Test: final accuracy = 0.455
```

- 은닉 계층 0 개인 단층 퍼셉트론을 기준으로 하여 다층 퍼셉트론의 적용 결과를 확인해보았다.

```
LEARNING_RATE=0.00022
hidden_config = [12, 6, 4]
steel_exec(epoch_count=50, report=10)
```

```
은닉 계층 3개를 갖는 다층 퍼셉트론이 작동되었습니다.
Epoch 10: loss=1.922, accuracy=0.345/0.353
Epoch 20: loss=1.683, accuracy=0.392/0.481
Epoch 30: loss=1.612, accuracy=0.426/0.478
Epoch 40: loss=1.597, accuracy=0.469/0.353
Epoch 50: loss=1.578, accuracy=0.461/0.501

Final Test: final accuracy = 0.501
```

- 은닉 계층 3 개를 갖는 다층 퍼셉트론을 적용하고, 학습률 및 퍼셉트론의 개수를 여러 번 조정하였지만 크게 향상된 정확도를 얻지 못했다. 또한, 단층 퍼셉트론에서의 하이퍼 파라미터를 조정하였을 때 더 높은 결과를 얻지 못하리라는 근거가 없다.
- 학습률을 잘 조정해주는 알고리즘을 적용해야 이런 점을 보완할 수 있으리라는 생각이 들었다.

결론

다층 퍼셉트론과 비선형 활성화 함수를 통하여 단층 퍼셉트론으로는 불가능한 비선형 결과를 도출할 수 있다는 것을 알게 되었으며, 선형적인 방법을 아무리 도입해도 결국 선형적인 결과만 나온다는 것을 알게 되었다. 이에 따라 다층 퍼셉트론을 아무리 적용해도 비선형 활성화 함수를 도입하지 않으면 결국 선형적인 결과가 나온다는 것이 흥미로웠고 그와 관련된 에피소드나 수학적 원리가 로 인해 이를 잘 이해할 수 있게 되었다.

너무 많은 퍼셉트론과 은닉 계층을 도입할 시에는 오히려 정확도가 떨어지거나 제대로 학습이 되지 않는 것을 확인할 수 있었는데, 이는 데이터의 양이 충분하지 않아서 발생하는 것으로 판단된다.

다층 퍼셉트론을 적용한 후 학습률을 조정하는 방식으로 정확도를 향상시키려고 노력하였으나, 데이터의 양이 다층 퍼셉트론을 적용하기에 충분하지 않았는지 정확도가 눈에 띄게 향상되지는 않았다. 많은 데이터를 가지고 있는 다층 퍼셉트론을 적용하여 눈에 띄는 향상도를 보이는 주제를 찾아서 수행해 봐야겠다는 생각이 들었다.