인공지능 3차 과제

2015871005

컴퓨터과학부

김도현

1. 코드

코드의 길이가 너무 길기때문에, 전체 함수와 코드에 대한 설명은 업로드한 소스코드에 주석으로 작성하였습니다. 아래는 코드의 효율을 높이기 위해 적용한 방법에 대해서만 작성하였습니다.

1) Lecun's backpropagation algorithm

backpropagation 학습 구현 방법 중 Lecun's backpropagation algorithm 을 cpp 코드로 구현하였습니다. Lecun's backpropagation의 구현은 각 노드에서 계산되는 delta weight값을 저장하는 과정이 필수적이므로, 이를 포함한 노드의 학습에 필요한 데이터들을 클래스의 변수로 설정하여 코드를 작성하였습니다.

2) 행렬곱을 이용한 레이어의 모든 노드의 output 계산

이 중 가중치 값은 레이어의 다음 노드의 net값을 계산할때, 행렬로 구현 하여, 병렬 연산하면 학습의 더 빨라진다는 사실을 바탕으로, weight 값들은 matrix라는 행렬의 곱 연산이 가능한 클래스를 구현하여 저장하였습니다.
아래는 Node 객체에 저장되는 변수들의 리스트 입니다.

3) 전체 변수를 관리하는 클래스의 작성.

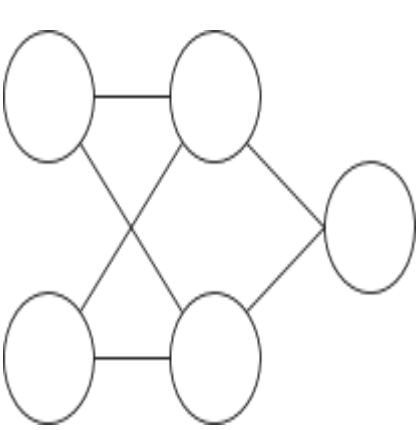
이번 과제의 요구 사항은, 레이어와 노드의 개수를 동적으로 받아야함이 조건이였습니다. Net클래스를 작성하여, 각 레이어의 노드의 수와, 레이어의 개수를 동적으로 할당 받고, 이를 학습할때는, NET 클래스의 함수로서 변수를 아웃풋 값을 만들고, 가중치를 수정했습니다.

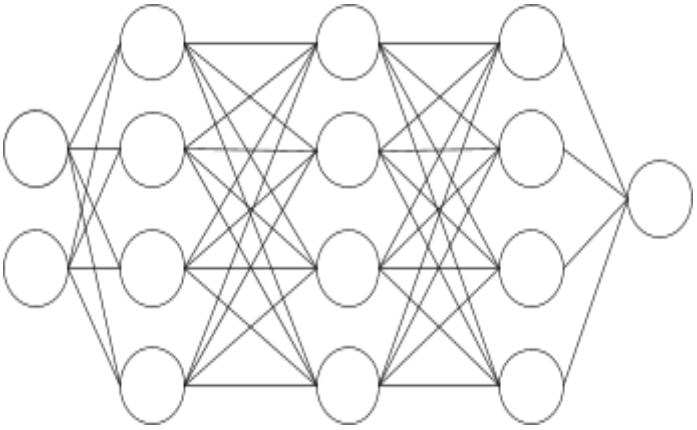
4) RMSE에러

에러 그래프를 그릴때, y축 변수를 하나로 통일하기 위해서, RMSE에러는 XOR 게이트에서 (0,0), (1,0), (0,1), (1,1)일 때의 에러의 제곱 합의 평균을 기준으로 보았습니다.

(DONUT, XOR, AND, OR 에 대한 실행 결과 스크린샷은 보고서 맨 마지막에 첨부하였습니다.)

2. learning rate, 활성함수, network 크기에 따른 성능 시험.

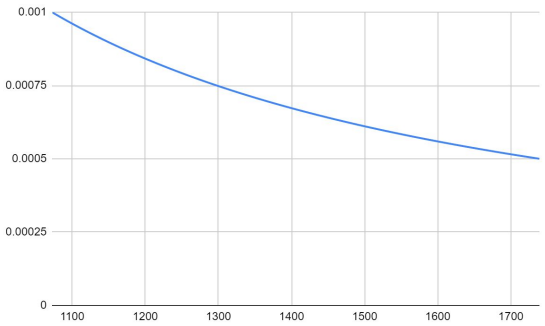




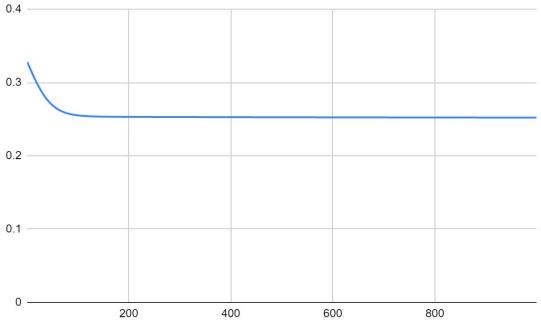
위는 아래 실험에서 사용할 다층퍼셉트론의 모델입니다. 간단한 모델에서는 인풋레이어 (2) – 히든레이어(2) – 아웃풋 레이어, 가장 간단한 다층 퍼셉트론 모델을 사용할 것이고, 성능 비교를 위해, 레이어의 개수, 노드의 개수를 늘린 모델은 오른쪽같이
인풋레이어(2)-히든레이어(4)-히든레이어(4)-히든레이어(4)-아웃풋레이어 로 생긴 모델을 사용할 것입니다.

실험 1. 간단한 모델, 활성함수 : sigmoid function

learning rate = 1, activation function 을 sigmoid function으로 두고 XOR 게이트 생성을 목적으로 코드를 실행한 결과, 아래와 같이 꾸준하게 에러가 감소하였고, XOR 학습이 잘 수행됨을 알 수 있었습니다.

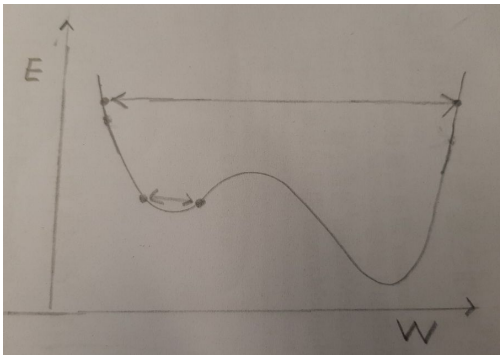


왼쪽 : y축 : RMSE , x축 시행횟수, 6000번 실행한 결과입니다.
오른쪽 : y축은 RMSE, x축은 시행횟수, rmse값이 0.001에서 0.005까지 감소할 때의 그래프입니다.



다른 수행 결과, 일반적으로 사용된다는 learning rate= 0.05 에서 학습이 진행되지 않음을 확인 할 수 있었습니다.

왼쪽은 Learning rate = 0.05 일 때 XOR 을 학습하는 과정의 에러 그래프 입니다. 학습이 진행되지 않음을 확인 할 수 있습니다.

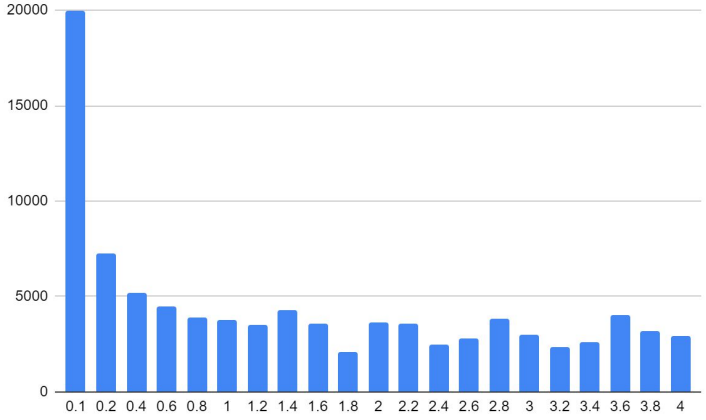


위의 원인을 gradient descent 과정에서 가중치의 변화량을 올바르게 설정하지 못했기 때문이라고 추측했습니다.
위의 경우는 learning rate 가 너무 적기 때문에 그림의 왼쪽 처럼 올바른 값을 찾지 못하는 경우라 볼 수 있습니다. 따라서 위 조건에서의 올바른 learning rate를 구하는 코드를 작성하여, learning rate 변화에 따른 코드의 성능을 확인 할 수 있었습니다.

<gradient descent : 가중치 변화에 따른 에러 함수>

3. 실험: 각 활성함수의 러닝레이트에 따른 성능 비교.

XOR 게이트를 찾는 과정에서, learning rate 를 0.1부터 0.2씩, 4까지 증가시켜 가면서, 학습이 가능한 learning rate를 찾는 과정을 100회 반복하여 학습이 가능한 learning rate를 찾아 볼 수 있었습니다. RMSE 에러가 0.0001이하 인 경우, 학습이 완성되었다 가정하고, 학습이 완성될때까지의 횟수를 측정하였습니다.



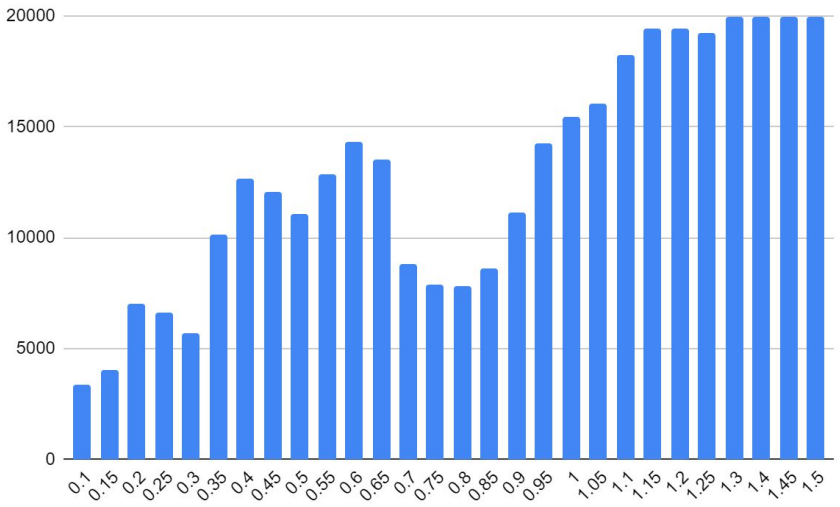
<x축: 러닝 레이트, y 축: 가중치 조정횟수의 평균>

활성함수 : sigmoid function
히든레이어의 개수 : 1개
히든레이어의 노드의 개수 : 2개

위의 실행의 탈출조건은 EPOCH > 20000으로 설정하였습니다. 따라서, 0.2 이하의 learning rate에서는 학습이 불가능한 경우가 많다는 것을 확인 할 수 있습니다. 위의 실험 데이터를 통해서, sigmoid function 으로 XOR 게이트를 구하는 과정은, 러닝 레이트가 0.5 이상에서 성능이 좋음을 확인 할 수 있었습니다.

위의 실험으로 sigmoid 함수에 대해 유효한 learning rate를 알아 볼 수 있었습니다.
같은 방식으로, leaky Relu를 활성함수로 썼을때의 결과도 확인 할 수 있었습니다.

3-2) leaky Relu function 의 learning rate 에 따른 성능 비교.



<x축: 러닝 레이트, y 축: 가중치 조정횟수의 평균>

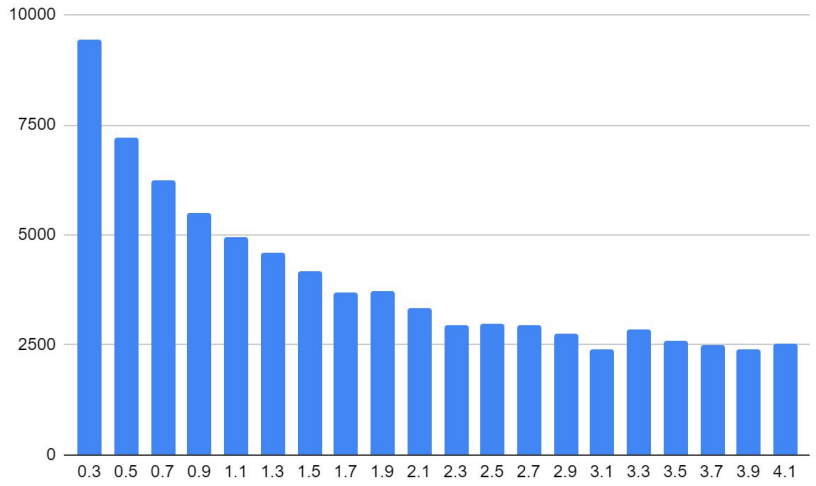
leaky Relu: x>0 y=x, y=0.2x, x<0
각 y축은 x축의 learning rate 에 대해 100회 시행한 평균입니다.

탈출 조건

1) RMSE < 0.0001
2) Epoch > 20000

위의 결과를 토대로, leaky relu 함수를 사용할 때, 0.1~0.3 의 sigmoid에서 추천된 값보다 작은 값을 사용해야 함을 확인 할 수 있습니다. 1.3을 넘어가면 학습 자체가 불가능 함도 확인 할 수 있었습니다. 시그모이드 보다 작은 값을 써야하는 이유는 상대적으로 leaky Relu 도함수의 크기가 sigmoid 보다 크기때문에, loss 함수에 대한 가중치의 gradient 값이 비슷하려면, leaky Relu 함수 실행에 곱해지는 learning rate 의 값이 상대적으로 작아야 합니다.

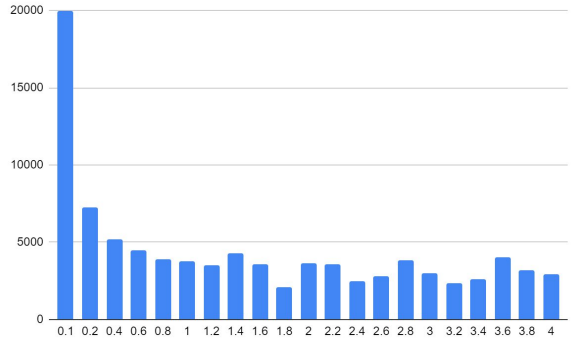
3-3) 노드의 개수, 레이어의 개수에 따른 효율



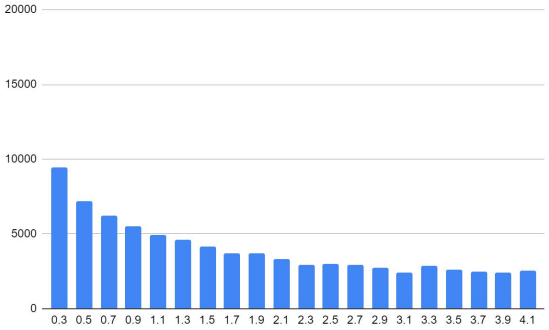
히든 레이어 2개, 각각 노드의 개수가 3개, 4개로 실행한 결과 입니다.
각 y축은 x축의 learning rate 에 대해 100회 시행한 평균입니다.

탈출 조건

3) RMSE < 0.0001
4) Epoch > 20000



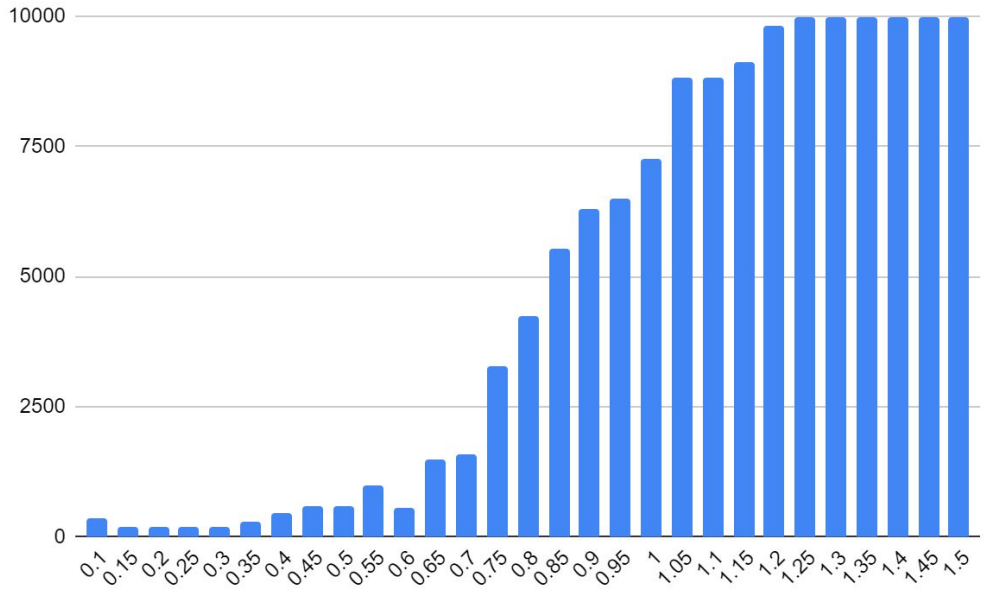
< 히든 레이어 1개, 노드 2개>



<히든레이어 2개, 노드 각각 3개, 4개>

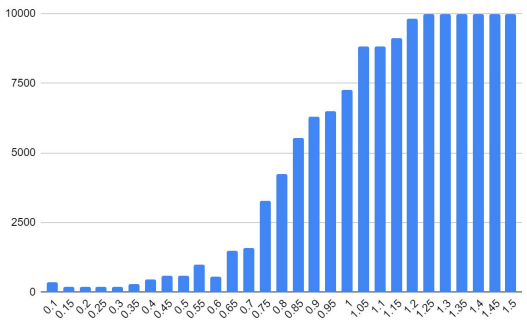
시그모이드 함수에서 실행했을 때, 큰 효율의 차이는 확인할 수 없었습니다.

leaky Reru, 노드,레이어 증가와 효율 증가의 관계.

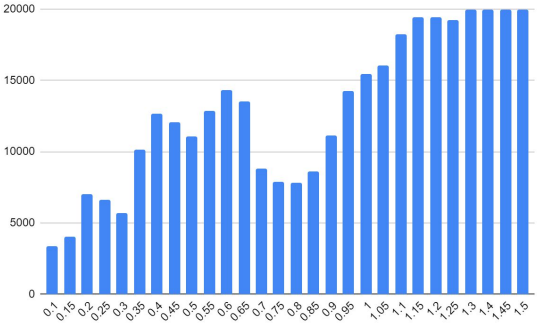


leaky Relu, 히든 레이어 3개, 각 노드의 수 4개로 실행
x축: learning rate, y축: 100시행한 평균
탈출조건 : 1)RMSE <0.0002 2) 10000번 시행.

위의 히든레이어 1개, 노드 갯수2 개의 sigmoid, leaky Relu의 실행비교에서는 큰 성능차이를 보일 수 없었으나, 레이어의 수를 증가시켰을때, 큰 성능 차이를 확인 할 수 있었습니다. 100번 시행한 평균으로 보았을때, 노드와 레이어의 수를 증가시킨 leaky Relu 모델에서는 평균 시행횟수가 learning rate = 0.15 일때, 177번, 0.2일때, 188번으로, 히든레이어 1개짜리 leaky layer 모델보다 성능이 10배 정도 향상된걸 확인 할 수 있었습니다.



1) 히든 레이어 3개 노드 4개



2) 히든레이어 1개, 노드 2개

결론

1) leaky Relu를 쓰는 것이 sigmoid 보다 효율이 좋습니다.
2) leaky Relu를 쓸 때, layer 와 뉴런 개수를 조절하면 효율은 효율에 큰 영향을 미칩니다.

,위의 조건 에서는 성능이 10배이상 차이가 났습니다.

3) 활성함수의 도함수 크기와 learning rate 는 반비례 합니다.

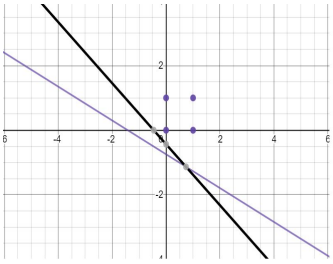
과제 수행물.

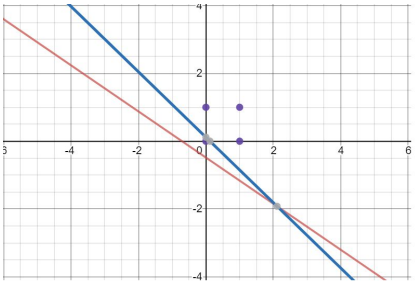
코드 수행결과 이외의 과제 수행 텍스트 파일은 폴더(텍스트폴더)에 따로 저장했습니다.

코드 수행 결과. 직선 이동.

XOR 실행결과.

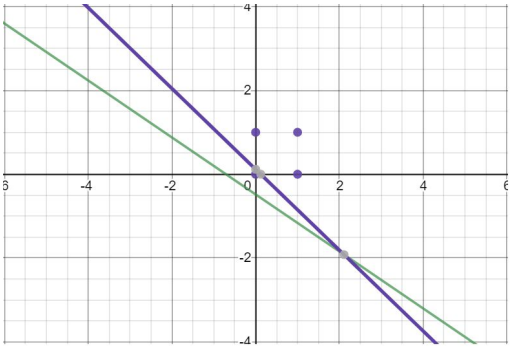
case 1 : activation function = sigmoid function, learning rate = 1.

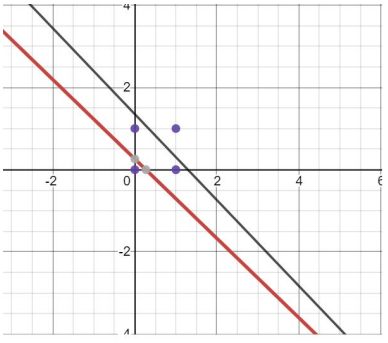




1) 왼쪽: 200회 반복, RMSE: 0.2203

2) 오른쪽: 300회 반복, RMSE ERROR : 0.1570



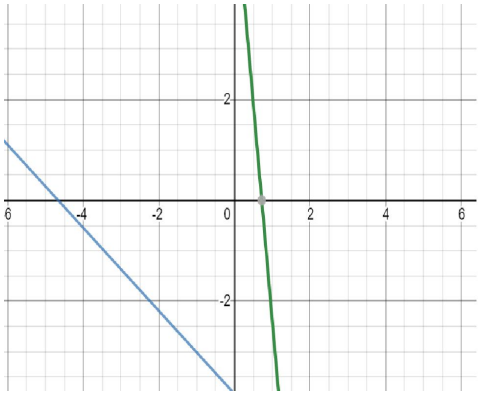


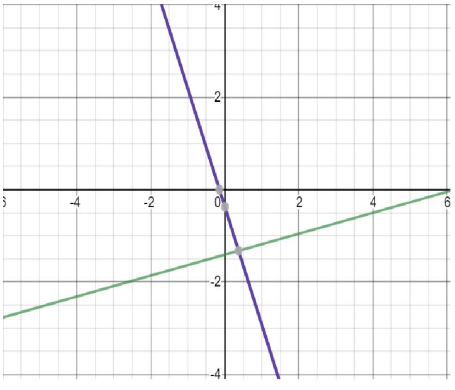
3)400번 반복, RMSE : 0.0325

4) 500회 반복, RMSE : 0.0291 학습 성공,

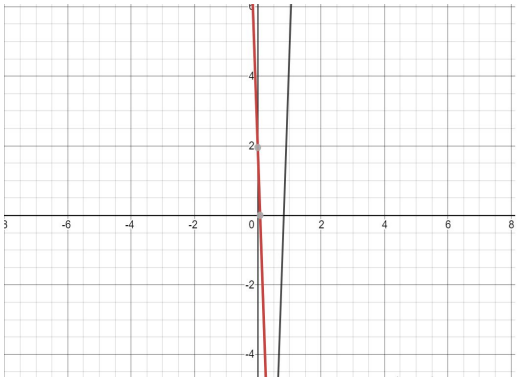
DONUT 실행결과.

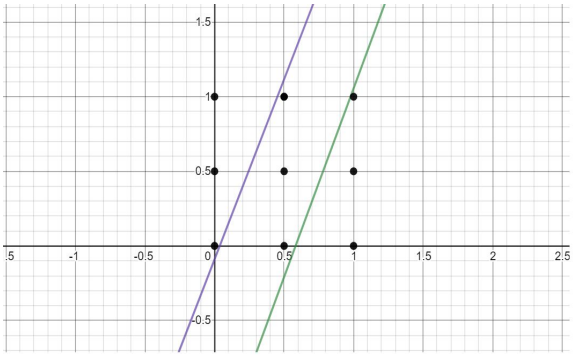
case 1 : activation function = sigmoid function, learning rate = 1.



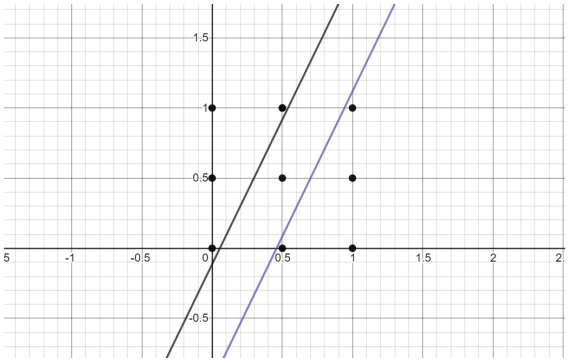


1) 왼쪽: 200 회 학습, RMSE ERROR : 1.005, 활성함수 : sigmoid function
2) 오른쪽 1100회 학습 , RMSE ERROR : 0.9991, 활성함수 : sigmoid function





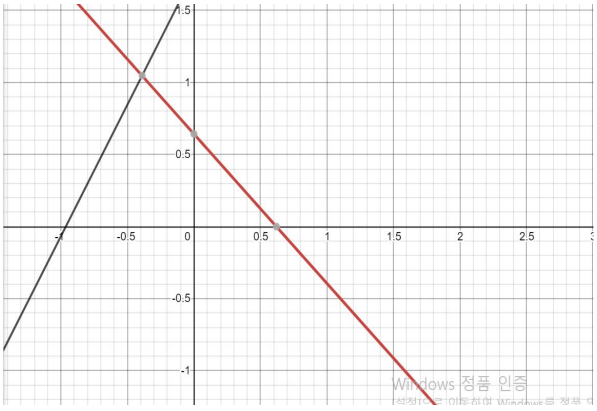
3) 왼쪽 : 3500회 학습 : RMSE ERROR : 0.0987, 활성함수 : sigmoid function
4) 오른쪽 : 4100 회 학습 : RMSE ERROR : 0.0581 , 활성함수 : sigmoid function

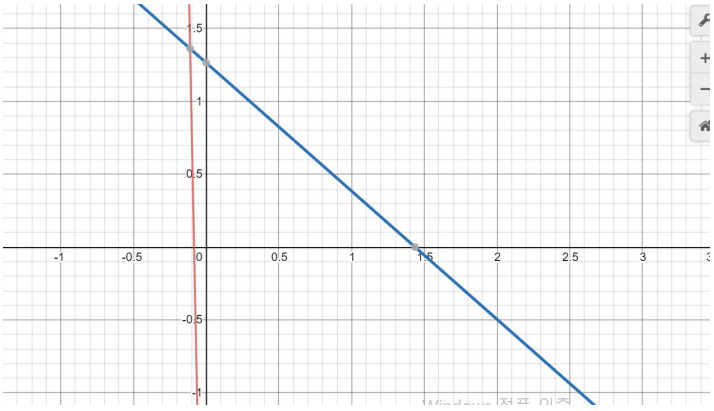


5) 4300회 학습 , RMSE ERROR : 0.0152, 학습 성공.

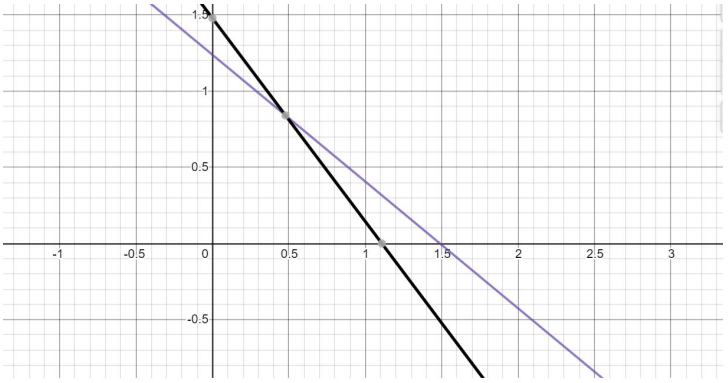
AND GATE

활성 함수 : sigmoid function





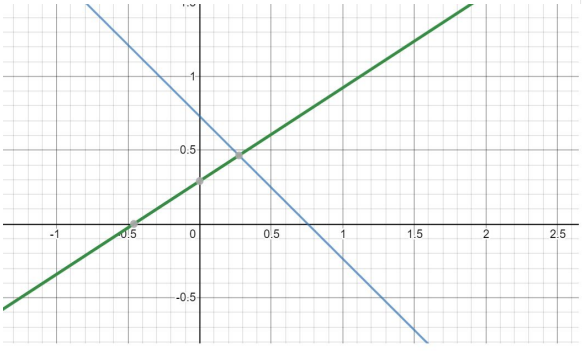
1) 왼쪽 : 0회 시행, RMSE ERROR : 0.395
2) 오른쪽 : 100회 시행, RMSE ERROR : 0.0094

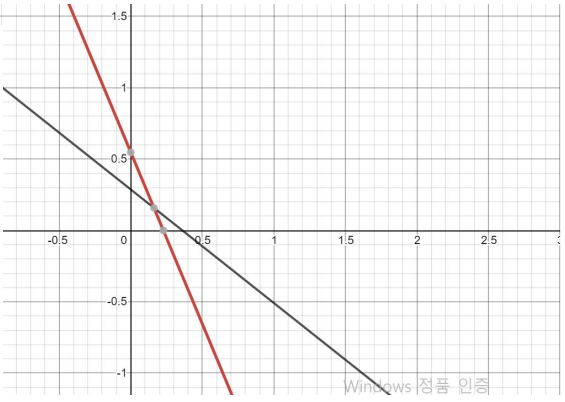


3) 200회 시행, RMSE ERROR : 0.0012, 학습 성공

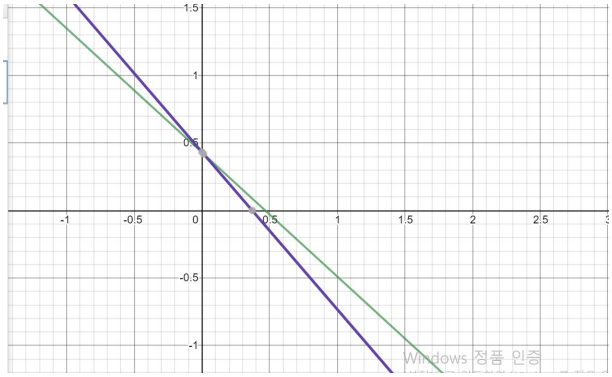
ORGATE

활성함수 : sigmoid function





1) 왼쪽: 0회 학습, RMSE ERROR : 0.20737
2) 오른쪽: 50회 학습 : RMSE ERROR : 0.0807



3) 200회 학습 : RMSE ERROR : 0.00167 학습 성공