인공지능 데이터 학습 보고서 #1

컴퓨터 과학부
2014920037 이정한

목차

뉴럴 네트워크의 핵심적인 기능인 학습, 예측 등의 기능을 network.py 내의 클래스에서 구현해두고 실제 실험은 main.py 에서 진행하는 방법으로 과제를 진행했습니다.

Network.py 설명

1. act 클래스
   - 활성화 함수와 미분함수를 모아둔 클래스입니다.

   A. relu

   B. sigmoid

   C. linear
2. LayerFile 클래스
   - 뉴럴 네트워크의 Weights값을 저장하고 불러오는 기능을 하는 클래스입니다.
   - NeuralNetwork 클래스에서 내부적으로 사용합니다.
3. NetData 클래스
   - 각종 학습데이터를 모아둔 클래스입니다.

   A. and_set

   B. or_set

   C. xor_set

   D. donut_set
   - 각각의 셋에 대하여 norm_set 메소드가 존재합니다. 게이트에 적합하게 정규분포로 데이터를 다량 생성합니다.
4. NeuralNetwork 클래스
   - Layer의 node수가 담긴 리스트를 바탕으로 뉴럴 네트워크를 관리합니다.
   - 데이터 셋은 한 row가 하나의 데이터 셋을 나타내도록 되어있습니다.
   - 아래 페이지에 클래스에서 만든 모든 메소드의 설명이 있습니다.
   - 에러함수는 Mean squared error를 사용했습니다.

   A. 생성자

      i. layerObject인자로 임의레이어를 생성하거나 파일이름 텍스트를 불러와서 네트워크를 생성합니다.

      ii. activation인자로 act클래스의 활성화 함수를 받습니다. (활성화 함수, 미분 활성화 함수)로 된 튜플을 받습니다.

      iii. train_x와 train_t로 데이터 셋을 받습니다.

      iv. last_sigmoid=True로 마지막 레이어의 활성화 함수를 sigmoid로 고정시킬 수 있습니다.

   B. train 메소드

      i. repeat만큼 네트워크를 학습시킵니다.

      ii. print_num만큼 중간 결과를 출력합니다.

      iii. show_error=True로 학습이 끝났을 때의 학습 중간의 error의 변화그래프을 보여줍니다.

      iv. show_line_num는 레이어가 (2,1)로 구성되어 있을 때, 결과를 가르는 직선의 학습을 보여 줄 때, 중간결과의 직선을 몇 개를 보여줄지를 정합니다. 0일경우 표시하지 않습니다.

      v. one_line, mini_batch는 각각의 bool값으로 트레인 셋을 한 셋씩 학습할지, 랜덤으로 고를지 정합니다. 둘다 false일 경우 전체 트레인 셋을 계속 학습합니다.

   C. test 메소드

      i. 해당 셋에 대한 정확도와 loss를 계산합니다.

      ii. show_w_plot=True면 직선으로 표현이 가능한 노드에 대해서 그래프를 표시합니다.

아래는 NeuralNetwork클래스의 정의입니다. 파이썬 쉘에서 help(NerualNetwork)로도 확인할 수 있습니다.

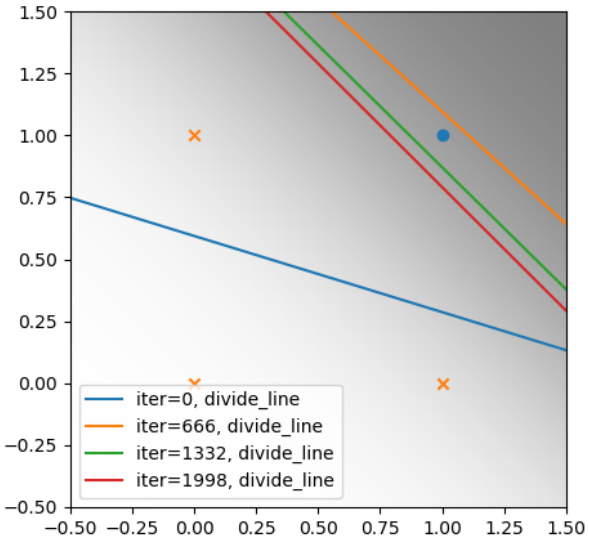
```
class NeuralNetwork(builtins.object)
 |  NeuralNetwork(layerObject, activation, train_x, train_t, last_sigmoid=False)
 |
 |  뉴럴 네트워크를 구현한 클래스입니다.
 |  x와 t의 데이터셋에서 한 행은 데이터셋, 한 열은 하나의 노드로 가는 입력값으로 인식합니다.
 |
 |  Methods defined here:
 |
 |  __init__(self, layerObject, activation, train_x, train_t, last_sigmoid=False)
 |      LayerObject가 문자열이면, 파일이름으로 인식하고 파일을 불러옵니다.
 |          그렇지 않으면 레이어의 리스트로 받고 노드들을 랜덤으로 초기화합니다.
 |      activation으로 (활성화함수, 미분활성화함수)를 받습니다.
 |      train_x, train_t로 필요한 데이터를 받습니다.
 |      classify가 True이면 에러함수를 cross_entropy함수를, False이면 Mean Squared함수를 사용합니다.
 |
 |  add_x0(self, array, axis=1)
 |      해당 행렬에 오른쪽 열 (axis=1,default) 혹은 아래쪽 행 (axis=0)에 1값을 추가합니다.
 |
 |  back_propagation(self, diff_loss, flow_net, flow_out, train_x)
 |      주어진 loss의 미분값과 중간결과값들로
 |      해당 network의 weights를 업데이트시킵니다.
 |
 |  create_weights(self)
 |      뉴럴 네트워크를 만들 때 w를 초기화합니다.
 |
 |  get_accuracy_loss(self, output, target)
 |      정확도와 loss를 반환합니다.
 |
 |  get_diff_loss(self, output, target)
 |      최종 에러에 대한 미분을 반환합니다.
 |
 |  predict(self, train_data=None)
 |      train_data로 예측합니다.
 |      최종 output, 중간net, 중간output리스트를 반환합니다.
 |
 |  save(self, filename)
 |      현재 네트워크의 weights 정보를 저장합니다.
 |
 |  show_error_plot(self, error_list, first_cut_off=True)
 |      error_list를 각각 (i,error)로 받아서
 |      그래프로 표시해줍니다.
 |      first_cut_off로 초반의 에러 표시를 버릴수 있습니다.
 |
 |  show_final_plot(self, w_list=None)
 |      최종 결과값에 대한 그래프를 표시.
 |      w_list가 있으면, 각각의 (i,w)에 대한 직선들을 표시합니다.
 |
 |  show_w_plot(self, flow_out, train_x, train_t)
 |      네트워크의 노드들 중에서 Input이 2인 노드들을
 |      각각 직선으로 표시합니다.
 |
 |  test(self, test_x=None, test_t=None, show_w_plot=False)
 |      test_x과 test_t의 정확도와 loss를 계산합니다.
 |      show_w_plot=True로 하면
 |      show_w_plot의 함수기능도 수행합니다.
 |
 |  train(self, learning_rate=0.1, repeat=100000, print_num=10, show_error=False, show_line_num=0, mini_batch=False,
one_line=False)
 |      학습을 시키면서 print_num만큼 중간 과정을 출력합니다.
 |      show_error로 error를 출력합니다.
 |      show_line_number에 1이상의 값을 주면,
 |      인풋2, 아웃풋1일때 결과를 나누는 직선의 변화를 출력합니다.
 |      값은 출력할 직선의 갯수를 의미합니다.
```

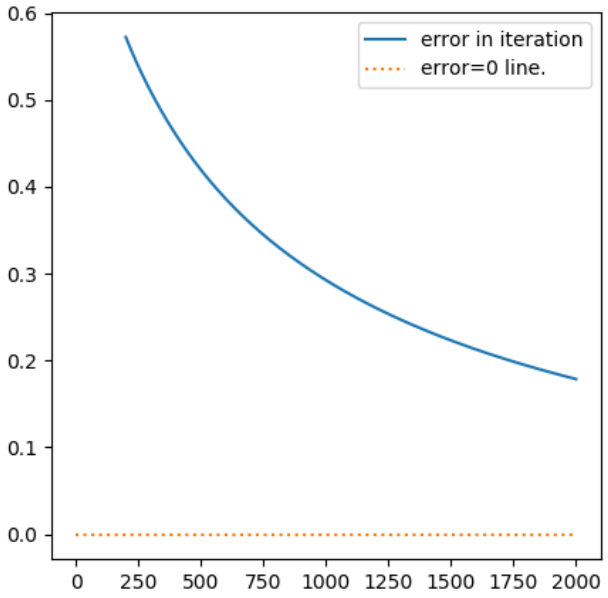
And 게이트 학습 실험

오른쪽과 같은 코드로 input:2, output:1인 노드를 시그모이드 활성화 함수를 적용해서 실험했습니다.

```
from network import act, NeuralNetwork, NetData

net = NeuralNetwork([2,1], act.sigmoid, *NetData.and_set)
net.train(repeat=2000, show_error=True, show_line_num=4)
net.save("net/and.txt")
```





왼쪽 그림은 반복을 통해서 뉴럴 네트워크가 and게이트로 학습되는 과정을 나타낸 그림입니다. 시간이 지나면서 초록색, 빨간색 직선이 and의 규칙에 맞게 노드를 분류하는 것을 볼 수 있습니다.

오른쪽 그림에서도 학습을 하면서 에러가 점점 줄어드는 것을 확인할 수 있습니다.

실제로 최종적으로 나온 w값을 확인해본 결과로는

w1 = 2.195398737927713

w2 = 2.2033509660080597

w0 = -3.438274907193242

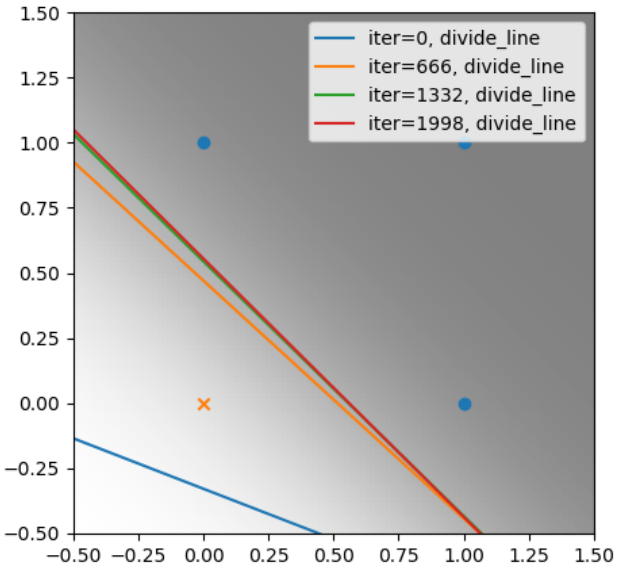
로 학습된 것을 확인했습니다.

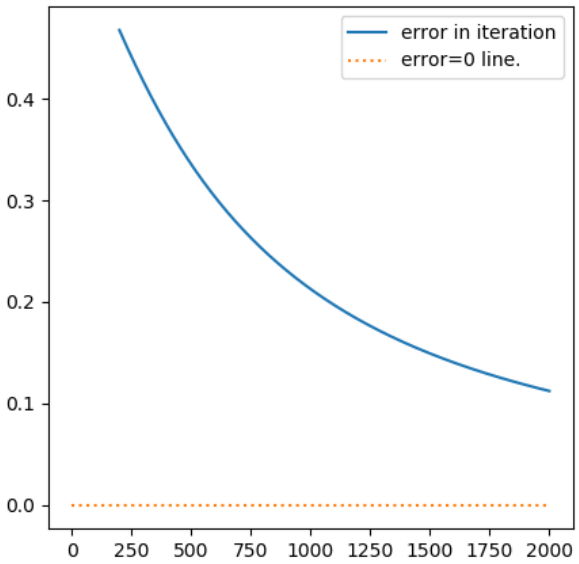
Or 게이트 학습 실험

And게이트와 같이 input:2, output:1의 레이어로 or게이트 데이터를 학습했습니다.

```
from network import act, NeuralNetwork, NetData

net = NeuralNetwork([2,1], act.sigmoid, *NetData.or_set)
net.train(repeat=2000, show_error=True, show_line_num=4)
net.save("net/or.txt")
```





And 게이트와 경우와 비슷하게 학습이 잘 되었습니다.

왼쪽 그림에서는 반복할수록 o 와 x 노드를 직선이 잘 나누도록 움직이고 있습니다.

오른쪽 그림에서도 반복할수록 에러가 줄어드는 것을 확인할 수 있습니다.

최종적 w 값은 다음과 같습니다.

w1 = 2.7741190405105924

w2 = 2.7993403907691508

w0 = -1.054443074115954

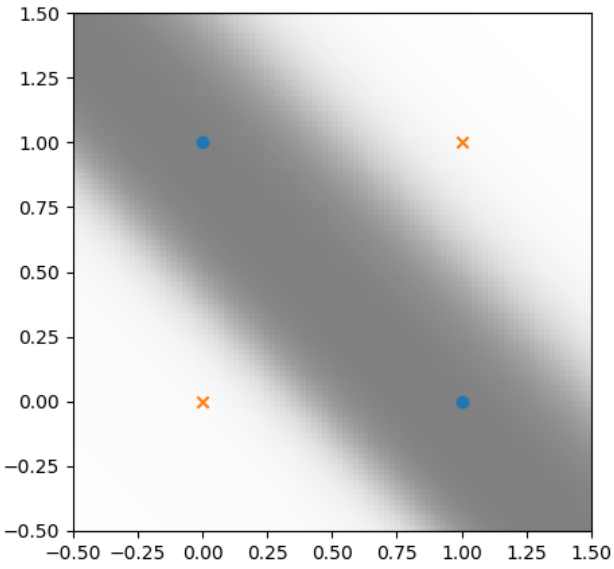
이때 and 게이트와 비교 했을 때 w1, w2 값은 비슷한데 비해서 w0 값이 달라졌다는 것을 알 수 있습니다.

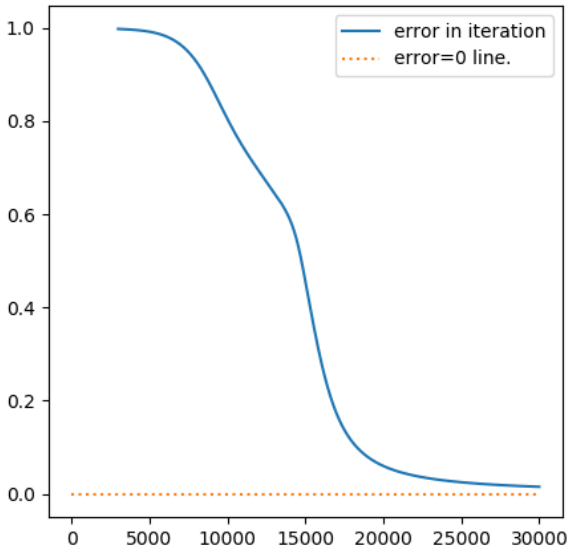
Xor 게이트 학습 실험

xor게이트는 Non linear 함수이기 때문에, 히든레이어가 필요하므로 input:2, hidden:2, output:1과 같은 네트워크로 구성했습니다. 또한 노드수가 많은 만큼 학습 반복도 늘렸습니다.

```
from network import act, NeuralNetwork, NetData

net = NeuralNetwork([2,2,1], act.sigmoid, *NetData.xor_set)
net.train(repeat=30000, show_error=True)
net.test(show_w_plot=True)
net.show_final_plot()
net.save("net/xor.txt")
```

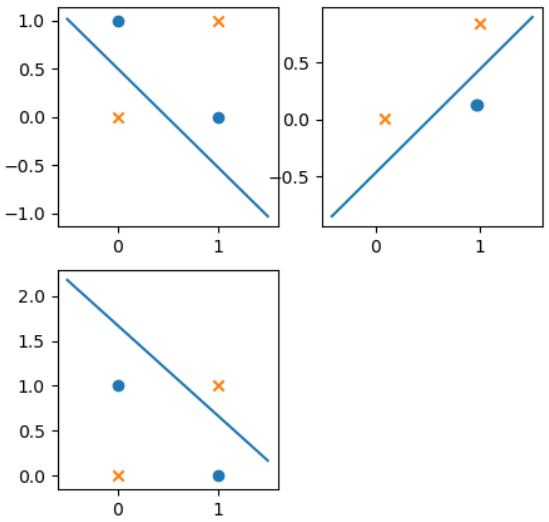




왼쪽 그림에서 검정색 성분이 True레이블에 가까운 값을 나타냅니다. 최종적으로 학습은 잘 되었습니다. 오른쪽 그림을 보면 15,000번 정도가 되야 에러가 많이 줄어들었습니다. 네트워크가 더 복잡해진 만큼 학습도 더 오래 걸리는 것으로 보입니다.

오른쪽 그림은 네트워크에서 노드들의 입력과 출력을 나타냅니다. 왼쪽의 2개의 그래프가 히든레이어의 노드 2개이며, 오른쪽의 그래프는 아웃풋 레이어의 노드를 나타냅니다.

아웃풋 레이어의 노드들은 히든 레이어를 거쳐서 원래 그래프의 위치와 다른 위치를 가지게 됨을 알 수 있습니다. 이런 방식으로 hidden layer를 이용하면 non-linear function을 만들 수 있습니다.

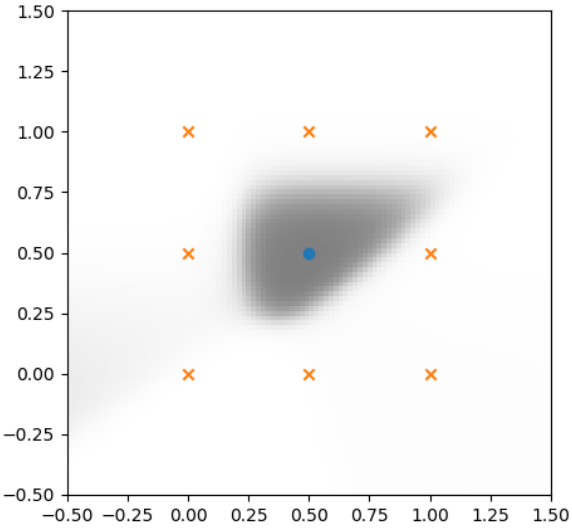


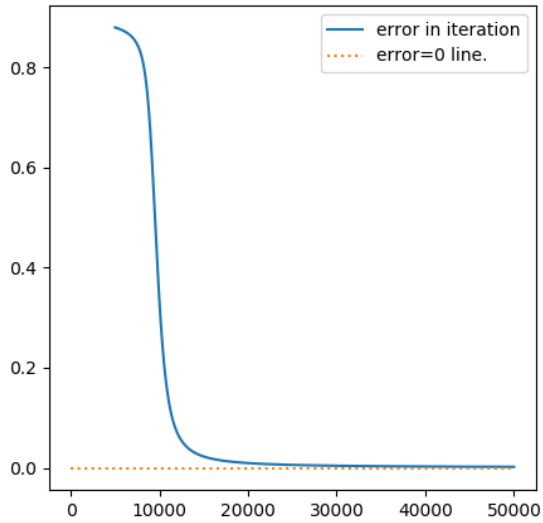
도넛모양 데이터 학습 실험

```
from network import act, NeuralNetwork, NetData

net = NeuralNetwork([2,10,1], act.sigmoid, *NetData.donut_set)
net.train(learning_rate=1.0, repeat=50000, show_error=True)
net.show_final_plot()
net.save("net/donut.txt")
```

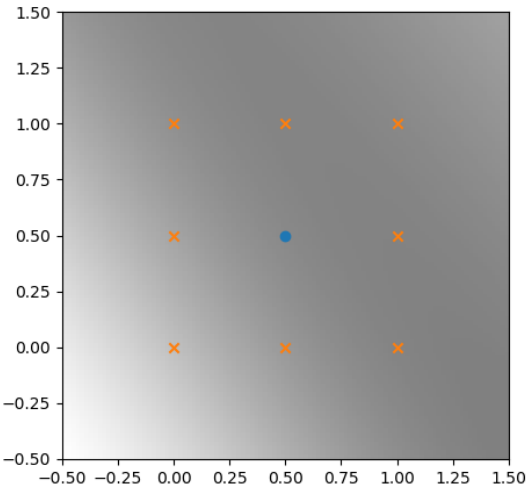
도넛 모양데이터를 학습 시키기 위해서 레이어 구성은 input:2, hidden:10, output:1 로 구성했습니다. 앞에서 기본값으로 learning_rate를 0.1로 한것과 다르게 1.0으로 한 이유는 학습속도 때문입니다. 이 상태에서는 learning_rate를 1.0까지 높여야만 20000번 이내로 충분한 학습이 이루어집니다. 그 이유는 데이터의 부족으로 추정됩니다. 현재 True레이블을 나타내는 데이터는 1개이기 때문에 이러한 데이터에 대한 학습이 충분히 이뤄지지 않는 것으로 보입니다.

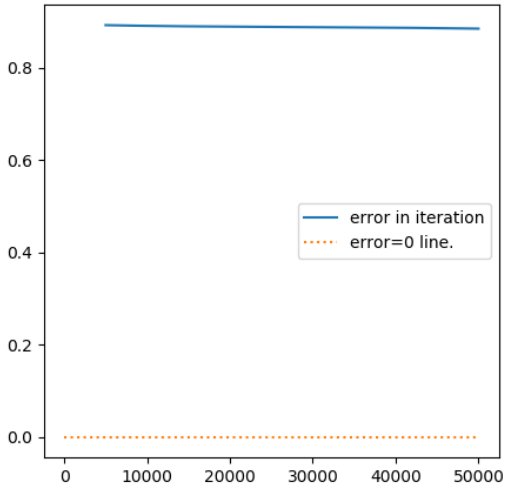




위와 같이 학습 되었으며 오른쪽 그래프를 보면 가운데 점만 잘 학습 된 것을 알 수 있습니다.

아래는 위의 코드에서 learning_rate=0.1 로 50000 번 학습한 결과입니다.



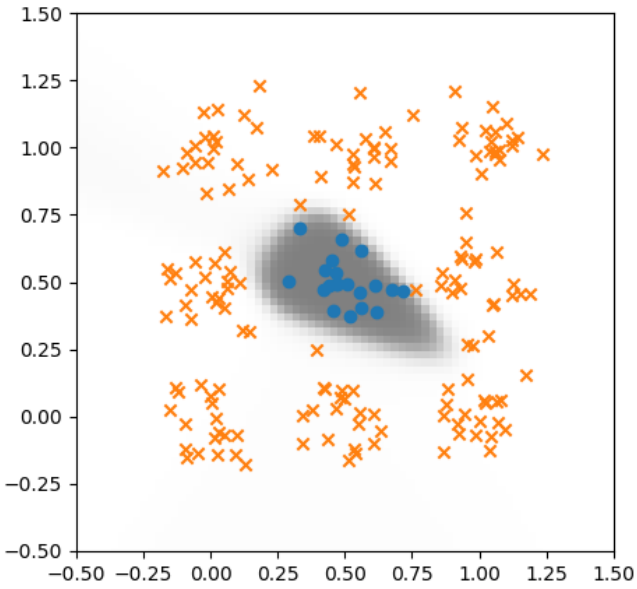


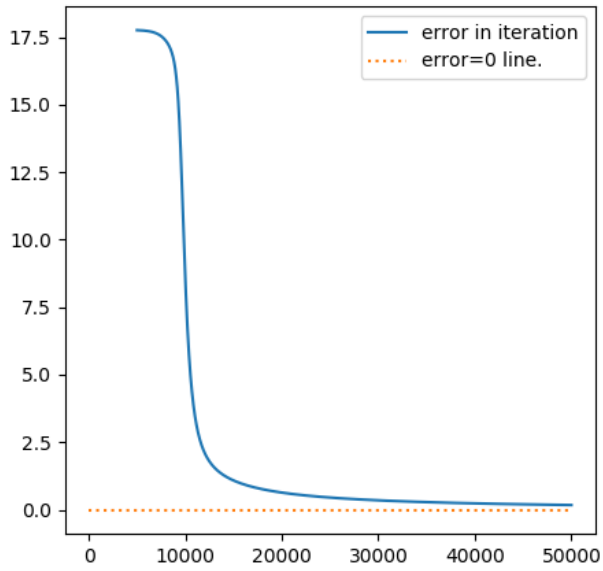
보시다시피 50000 번으로는 전혀 학습이 되지 않은 것을 알 수 있습니다.

이번에는 donut 데이터에 기반해서 정규분포로 여러 데이터셋을 생성해서 학습시켜 보겠습니다. 처음 donut 네트워크와 동일하게 하고 learning_rate 는 0.1 을 할당해도 학습할 수 있습니다.

```
from network import act, NeuralNetwork, NetData

net = NeuralNetwork([2,10,1], act.sigmoid, *NetData.donut_norm_set())
net.train(learning_rate=0.1, repeat=50000, show_error=True)
net.show_final_plot()
net.save("net/donut_norm.txt")
```





위와 같이 데이터 셋의 개수가 충분하면 learning_rate 조절없이 학습이 잘 되는 것을 확인할 수 있습니다.

relu와 sigmoid의 비교

relu활성화 함수의 결과는 0부터 무한대 사이이므로 확률의 결과인 0~1의 값으로 만들어주는 sigmoid나 tanh 함수등을 사용합니다. output의 활성화 함수는 시그모이드를 사용하고 히든레이어의 활성화 함수를 relu와 sigmoid를 사용했을 때를 비교해보겠습니다. 다음은 and 데이터 정규분포 셋을 미니배치를 사용한 relu, 미니배치를 사용하지 않은 sigmoid로 학습속도를 비교한 것입니다.

오른쪽이 작성한 코드입니다.

```
from network import act, NeuralNetwork, NetData

def func(actor, mini_batch):
    for i in range(5):
        net = NeuralNetwork([2,6,1], actor, *NetData.and_norm_set(), last_sigmoid=True)
        net.train(learning_rate=0.1,     repeat=200,     print_num=1, mini_batch=mini_batch)
        print()

print("relu-minibatch")
func(act.relu, True)

print("sigmoid-non-minibatch")
func(act.sigmoid, False)
```

```
relu-minibatch
Step 0 : Accuracy = 0.25000000, Loss = 35.50029288
Step 200 : Accuracy = 1.00000000, Loss = 1.11591936

Step 0 : Accuracy = 0.25000000, Loss = 51.08461998
Step 200 : Accuracy = 1.00000000, Loss = 0.55698302

Step 0 : Accuracy = 0.25000000, Loss = 58.80688804
Step 200 : Accuracy = 1.00000000, Loss = 1.48700222

Step 0 : Accuracy = 0.25000000, Loss = 54.33215437
Step 200 : Accuracy = 1.00000000, Loss = 0.71758983

Step 0 : Accuracy = 0.25000000, Loss = 33.81928019
Step 200 : Accuracy = 1.00000000, Loss = 0.61948806

sigmoid-non-minibatch
Step 0 : Accuracy = 0.25000000, Loss = 55.43896193
Step 200 : Accuracy = 0.91250000, Loss = 8.63658027

Step 0 : Accuracy = 0.25000000, Loss = 51.98081697
Step 200 : Accuracy = 0.88750000, Loss = 8.44212624

Step 0 : Accuracy = 0.25000000, Loss = 47.99845690
Step 200 : Accuracy = 0.75000000, Loss = 11.90582449

Step 0 : Accuracy = 0.25000000, Loss = 49.91683912
Step 200 : Accuracy = 1.00000000, Loss = 5.33764276

Step 0 : Accuracy = 0.25000000, Loss = 53.40956947
Step 200 : Accuracy = 0.92500000, Loss = 8.07695089
```

왼쪽은 위의 코드를 실행한 결과입니다.

relu활성화 함수를 사용했을 때가 sigmoid 함수를 사용했을 때보다 최종 에러가 현저하게 작은 것을 확인할 수 있습니다. 이는 relu가 sigmoid 함수 보다 학습속도가 빠르다는 것을 알 수 있습니다.

하지만 relu활성화 함수는 미니 배치에 따라서 학습의 진전이 매우 다릅니다. 오른 쪽과 코드로 xor데이터의 정규분포 데이터 셋을 10번 학습해보았습니다.

```
from network import act, NeuralNetwork, NetData

def func(actor, mini_batch):
    for i in range(10):
        net = NeuralNetwork([2,2,1], actor, *NetData.xor_norm_set(), last_sigmoid=True)
        net.train(learning_rate=0.1, repeat=1500, print_num=1, mini_batch=mini_batch)
        print()

print("relu-minibatch")
func(act.relu, True)
```

```
relu-minibatch
Step 0 : Accuracy = 0.50000000, Loss = 22.70924957
Step 1500 : Accuracy = 1.00000000, Loss = 0.13591232

Step 0 : Accuracy = 0.50000000, Loss = 22.99748680
Step 1500 : Accuracy = 0.52500000, Loss = 19.99462459

Step 0 : Accuracy = 0.50000000, Loss = 25.14083493
Step 1500 : Accuracy = 1.00000000, Loss = 0.15423804

Step 0 : Accuracy = 0.50000000, Loss = 30.58703976
Step 1500 : Accuracy = 0.48750000, Loss = 20.00163225

Step 0 : Accuracy = 0.50000000, Loss = 25.42751419
Step 1500 : Accuracy = 0.50000000, Loss = 20.00507284

Step 0 : Accuracy = 0.50000000, Loss = 31.23635214
Step 1500 : Accuracy = 0.50000000, Loss = 20.00015086

Step 0 : Accuracy = 0.50000000, Loss = 24.84395128
Step 1500 : Accuracy = 0.50000000, Loss = 20.04113688

Step 0 : Accuracy = 0.50000000, Loss = 27.57102461
Step 1500 : Accuracy = 1.00000000, Loss = 1.06083797

Step 0 : Accuracy = 0.50000000, Loss = 23.78839186
Step 1500 : Accuracy = 1.00000000, Loss = 0.16120719

Step 0 : Accuracy = 0.50000000, Loss = 28.40836010
Step 1500 : Accuracy = 0.50000000, Loss = 20.14141721
```

왼쪽이 그 결과입니다. Loss의 양상을 보면 진행에 따라서 매우 다른 것을 확인할 수 있습니다. 데이터 셋에서 랜덤으로 배치되는 방법에 따라서 학습의 양상이 달라지고 local minimum에 쉽게 빠지는 것으로 추측됩니다.