인공지능 Multi\_Layer Perceptron 구현

컴퓨터과학부 2014920044 이홍석

1. **프로그램 개요**

이번 과제에서 Numpy를 이용하여 Multi\_Layer Perceptron을 구현하였습니다. 각 레이어마다노드 수는 조절이 가능하게 되어있고 레이어의 수도 조절이 가능하게 되어있습니다. 각 노드마다 x0는 1로 고정되어 있고 w0+x1\*w1+x2\*w2+…을 이용하여 net을 구했습니다. 경사하강법을 이용하였고 weights의 initial value는 평균이 0, 표준편차가 1인 정규분포에서 뽑았습니다. Learning rate은 0.2이고 횟수는 어떤 input이냐에 따라 적당한 값이 나오도록 제가 조정하였습니다. Random의 seed값은 매번 학습 결과가 잘 나오도록 조정하였고 아래 결과에 사용한 seed를 각각 적어두었습니다. Activation function은 sigmoid(lambda=1)와 relu(f`(0)=0.5)를 사용하였습니다. 이 보고서에서 Error의 계산은 dataset을 돌고 나온 Error의 평균을 사용하였습니다. 사용환경은 Python 3.6.5 Anaconda를 사용하였고 IDE로 VSCode와 Jupyter notebook을 사용하였습니다. 사용한 모듈은 numpy와 matplotlib.pyplot, math를 사용하였습니다.

1. **코드 설명**
   1. 모듈선언

|  |
| --- |
| from numpy import array, dot, random  import numpy as np  import matplotlib.pyplot as plt  import math as mt |

numpy와 matplotlib.pyplot, math를 import 합니다.

* 1. Activation Function

|  |
| --- |
| sig = lambda x : 1.0/(1.0+mt.exp(-x\*1.0))  def relu(x):  if x > 0.0:  return x  elif x == 0.0:  return 0.5\*x  else:  return 0.0  # return 0.1\*x # Leaky ReLU  dif\_sig = lambda x : sig(x)\*(1.0-sig(x))  def dif\_relu(x):  if x > 0.0:  return 1.0  elif x == 0.0:  return 0.5  else:  return 0.0  # return 0.1 # Leaky ReLU  func = relu  dif\_func = dif\_relu |

Sigmoid를 lambda를 통해 구현했고 ReLU를 함수를 통해 구현하였습니다. ReLU는 x가 0일 때 기울기가 없으므로 임의로 0.5로 정했습니다.

Sigmoid함수를 미분한 함수를 dif\_sig로 정의하였습니다. ReLU함수를 미분한 함수도 정의하였습니다. ReLU같은 경우 주석처리한 부분을 사용하면 Leaky ReLU를 사용할 수 있습니다. func와 dif\_func에 원하는 함수명을 입력하면 해당 함수를 사용할 수 있습니다.

|  |
| --- |
| random.seed(seed=20181114) |

* 1. Random Seed와 InputData

seed값은 20181114 혹은 20181115를 사용하였습니다.

* 1. Input Data

|  |
| --- |
| # OR  '''data = [  (array([1,0,0]),0),  (array([1,0,1]),1),  (array([1,1,0]),1),  (array([1,1,1]),1)  ]'''  # AND  '''data = [  (array([1,0,0]),0),  (array([1,0,1]),0),  (array([1,1,0]),0),  (array([1,1,1]),1)  ]'''  # XOR  '''data = [  (array([1,0,0]),0),  (array([1,0,1]),1),  (array([1,1,0]),1),  (array([1,1,1]),0)  ]'''  # 도넛모양  data = [  (array([1,0,0]),0),  (array([1,0,0.5]),0),  (array([1,0,1]),0),  (array([1,0.5,0]),0),  (array([1,0.5,0.5]),1),  (array([1,0.5,1]),0),  (array([1,1,0]),0),  (array([1,1,0.5]),0),  (array([1,1,1]),0)  ] |

실행할 실험에 해당하는 InputData를 활성화하고 나머지는 주석처리합니다.

배열의 왼쪽은 x0, x1, x2를 의미하고 오른쪽은 y값을 의미합니다.

* 1. Graph 초기화

|  |
| --- |
| train\_set0\_x1=[]  train\_set0\_x2=[]  train\_set1\_x1=[]  train\_set1\_x2=[]  for a,b in data:  if b==0:  train\_set0\_x1.append(a[1:][0])  train\_set0\_x2.append(a[1:][1])  else:  train\_set1\_x1.append(a[1:][0])  train\_set1\_x2.append(a[1:][1])  fig = plt.figure()  subplot = fig.add\_subplot(2,2,2)  subplot.set\_xlim(-0.5,1.5)  subplot.set\_ylim(-0.5,1.5)  subplot.scatter(train\_set0\_x1, train\_set0\_x2, marker = 'x')  subplot.scatter(train\_set1\_x1, train\_set1\_x2, marker = 'o')  linex = np.linspace(-1,30,3) |

Graph를 위한 초기화 코드입니다. dataset에서 x와 y를 가져오고 y가 0일때와 1일때로 구분합니다. 그리고 x1과 x2로 구분하고 그것을 통해 subplot에 점을 찍습니다.

직선의 x는 -1부터 3까지 30개의 값으로 구성되어 있습니다.

* 1. Initialize

|  |
| --- |
| input node = [2] # Input Lyaer의 노드 수  hidden\_node = [2] # Hidden Layer의 노드 수 배열([2,3,5,4,2]와 같이 가능)  output\_node = [1] # Output Layer의 노드 수  layer = input\_node + hidden\_node + output\_node # 전체 Layer의 노드 수 배열  delta = [] # delta 값들을 저장할 배열  net = [] # net 값들을 저장할 배열  w=[] # weight 값들을 저장할 배열  c = 0.2 # Learning Rate  n = 100 # Training Iteration 횟수  # Initialize  for i in range(1,len(layer)):  temp = [] # weight 배열을 위한 temp  temp\_delta = [] # delta 배열을 위한 temp  temp\_net = [] # net 배열을 위한 temp  for j in range(layer[i]):  # 각 Layer의 노드 수 만큼 weight 생성  temp.append(random.normal(0.0,1.0,size=layer[i-1]+1))  temp\_delta.append(0) # 각 Layer의 노드 수 만큼 delta 생성  temp\_net.append(0) # 각 Layer의 노드 수 만큼 net 생성  w.append(temp)  delta.append(temp\_delta)  net.append(temp\_net) |

초기화를 진행하는 부분입니다. 각각 부분은 주석의 설명과 같습니다.

* 1. Training

|  |
| --- |
| # Training  error = []  for N in range(1,n+1): # 학습은 n번 진행  if N % (n/5) == 0: # 5개의 지점에서 출력  print('\nn=%d' % N)  ##### Layer 1 Node들의 직선 그래프 코드  for i in range(len(w[0])):  liney = -(w[0][i][1]\*linex+w[0][i][0]-0.5)/(w[0][i][2]+0.00000000001)  subplot.plot(linex,liney, label = 'L1 x%d, N = %d' % (i+1,N))  plt.legend(loc=2)  #####  temp\_error = 0 # dataset의 Error를 모으기위한 temp  # data에서 x,y를 가져옴  for x,y in data:  x = [x]  y = np.array([y])  # 각 노드마다 f(net)의 값을 x에 저장  for i in range(0,len(layer)-1):  temp = np.array(1.0)  for j in range(0,layer[i+1]):  net[i][j] = dot(x[i],w[i][j])  fx = func(net[i][j]) # f(x)  temp = np.append(temp,fx)  x.append(temp)    # Output Layer의 Node들의 delta값들을 구함  for i in range(layer[-1]):  delta[-1][i] = -(y[i]-x[-1][i+1])\*dif\_func(net[-1][i])  # Output Layer 이전 Layer부터 Layer1까지의 delta값들을 구함  for i in range(len(layer)-2,0,-1):  for j in range(layer[i]):  \_sum = 0  for k in range(layer[i+1]):  \_sum = \_sum + delta[i][k]\*w[i][k][j+1]  delta[i-1][j] = dif\_func(net[i-1][j])\*\_sum    # Output Layer부터 Layer1까지의 Weight들을 갱신함  for i in range(len(w)-1,-1,-1):  for j in range(len(w[i])):  for k in range(len(w[i][j])):  dw = -c\*delta[i][j]\*x[i][k]  w[i][j][k] += dw  # 5의 지점에서 출력  if N % (n/5) == 0:  print('x1=%f, x2=%f, y=%f, fx=%f, Error=%f' % (x[0][1],x[0][2],y[0],x[len(x)-1][1],(1/2)\*((y[0]-x[len(x)-1][1])\*\*2)))  temp\_error+=(1/2)\*((y[0]-x[len(x)-1][1])\*\*2)  error.append(temp\_error/len(data)) |

Training도 마찬가지로 설명을 주석에 적어 두었습니다. 중간에 학습하면서 그 과정을 직선으로 남기는 코드가 있습니다. 전체 학습을 5개의 체크포인트로 나누고 그 시기마다 직선을 그려서 학습과정을 보여줄 수 있습니다.

* 1. Error 그래프

|  |
| --- |
| subplot\_error = fig.add\_subplot(2,2,1)  line\_x = np.linspace(0,n,10000)  subplot\_error.plot(error) |

n번을 돌리는 동안 구했던 error를 그래프로 그립니다.

* 1. 도넛 모양 그래프

|  |
| --- |
| def calc(x):  x = [x]  # 각 노드마다 f(net)의 값을 x에 저장  for i in range(0,len(layer)-1):  temp = np.array(1.0)  for j in range(0,layer[i+1]):  net[i][j] = dot(x[i],w[i][j])  fx = func(net[i][j]) # f(x)  temp = np.append(temp,fx)  x.append(temp)  return x[-1][1]  subplot1 = fig.add\_subplot(2,2,3)  subplot1.set\_xlim(-0.5,1.5)  subplot1.set\_ylim(-0.5,1.5)  subplot1.scatter(train\_set0\_x1, train\_set0\_x2, marker = 'x')  subplot1.scatter(train\_set1\_x1, train\_set1\_x2, marker = 'o')  # x 값을 구함  locations = []  for x2 in np.linspace(-0.5, 1.5, 100):  for x1 in np.linspace(-0.5, 1.5, 100):  locations.append(np.array([1,x1,x2]))  # x에 대한 y값을 구함  p\_vals = []  for x in locations:  p\_vals.append(calc(x))  p\_vals = np.array(p\_vals)  p\_vals = p\_vals.reshape((100,100))  # 그림을 그림  subplot1.imshow(p\_vals, origin = 'lower', extent = (-0.5,1.5,-0.5,1.5),cmap=plt.cm.gray\_r, alpha=0.5) |

도넛 모양의 그림은 학습이 끝난 Neural network에 x값들을 넣어서 그 결과값을 무수히 점을 찍어서 그림으로 나타내는 것입니다. 따라서 도넛이 구름처럼 나타날 것입니다.

* 1. 파일 출력 & 그래프 show

|  |
| --- |
| def filewrite(weight):  f = open('weights.txt','w')  f.write('[')  for i in range(len(w)):  f.write('[')  for j in range(len(w[i])):  f.write('[\n')  for k in range(len(w[i][j])):  f.write('%10f\n' % w[i][j][k])  f.write(']\n')  f.write(']')  f.write(']')  f.close()  filewrite(w)  plt.show() |

weight들을 배열로 출력하는 코드입니다. weights.txt에 결과를 저장합니다.

만든 그래프들을 GUI로 출력합니다.

1. **실험 결과**
   1. AND, OR, XOR 구분실험
      1. **AND Gate**
         1. **ReLU 사용**

Learning rate은 0.2, Training 횟수는 50번을 적용하였습니다. Seed는 ‘20181115’를 적용하였습니다. Hidden Layer의 수는 0이고 Input Layer의 노드 수는 2, Output Layer의 노드 수는 1로 적용하였습니다. AND에서 ReLU를 적용할 경우 net이 음수가 되는 경우가 존재하고 그 결과 기울기가 0, f(net)값이 0이 되므로 학습이 안되는 경우가 많았습니다. 따라서 ReLU보다는 Leaky ReLU를 사용하여 음수일때도 학습이 되도록 만들어야 합니다. 또는 Sigmoid나 다른 음수를 처리할 수 있는 Activation Function을 사용해야 합니다.

그림 1과 같이 Error는 쭉 감소하였습니다.

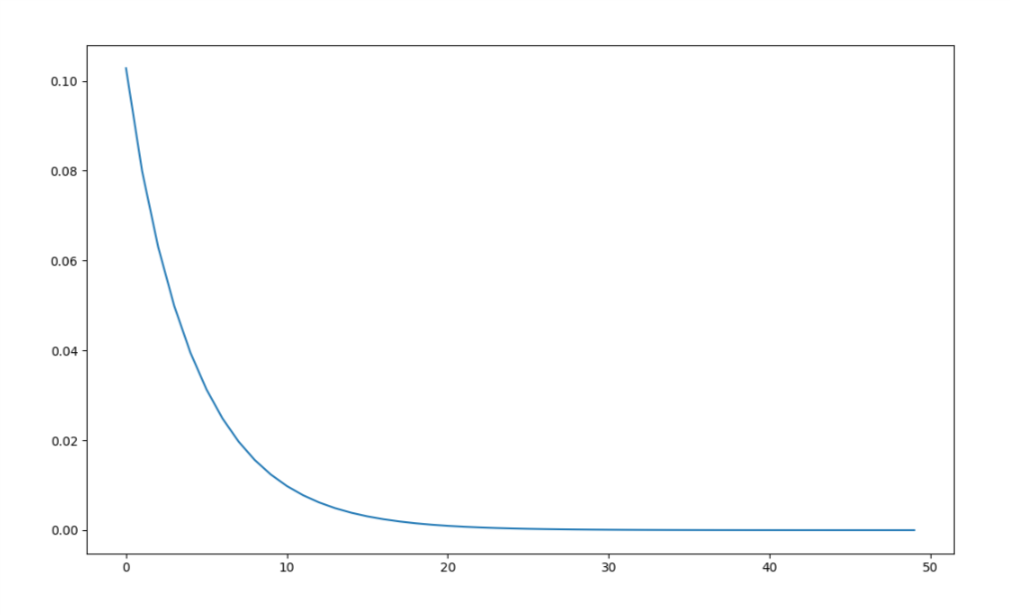


그림 1 : Error(AND, ReLU)

그림 2 : Linear Graph(AND, ReLU, Node(x1))

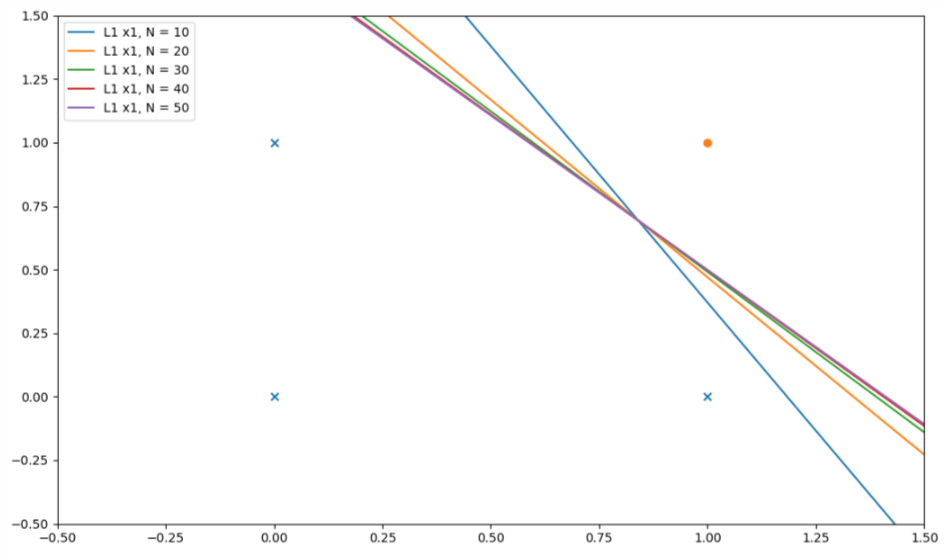


그림 2와 같이 그래프가 파란 선에서 보라 선으로 이동하는 것을 볼 수 있습니다. 또, Output Layer의 노드인 x1에서의 그래프 모양은 위 그림 2와 같다는 것을 알 수 있습니다.

* + - 1. **Sigmoid 사용**

이번에는 Training 횟수를 1000번으로 적용하였습니다. 나머지 조건은 위의 ReLU와 같은 조건을 사용하였습니다. 이 경우에는 Training 횟수가 많이 늘어난 것을 알 수 있습니다. ReLU보다는 학습속도가 느리지만 ReLU를 사용했을 때 음수에서 학습이 안되는 것을 보완할 수 있습니다.

ReLU와 마찬가지로 학습을 진행할수록 Error가 감소하는 것을 볼 수 있습니다.

그림 3 : Error(AND, Sigmoid)

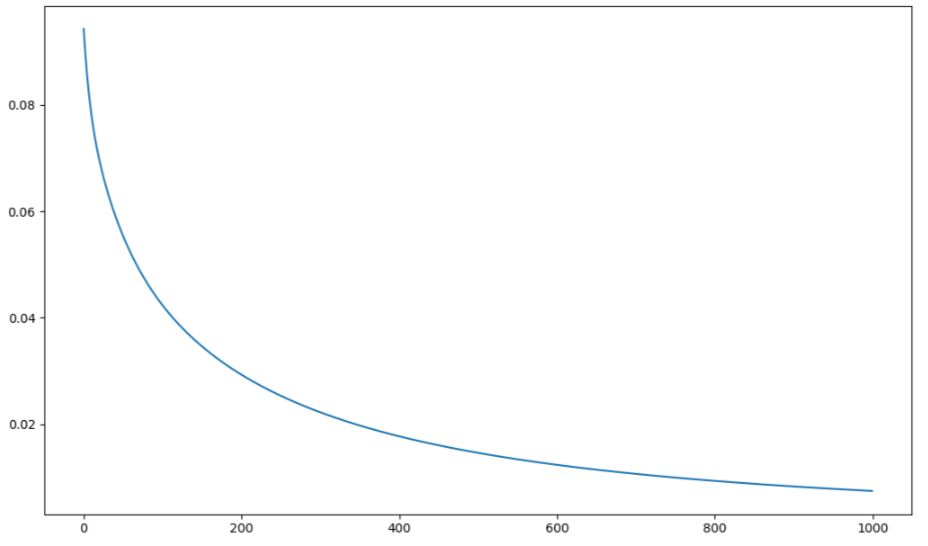


그림 4 : Linear Graph(AND, Sigmoid, Node(x1))

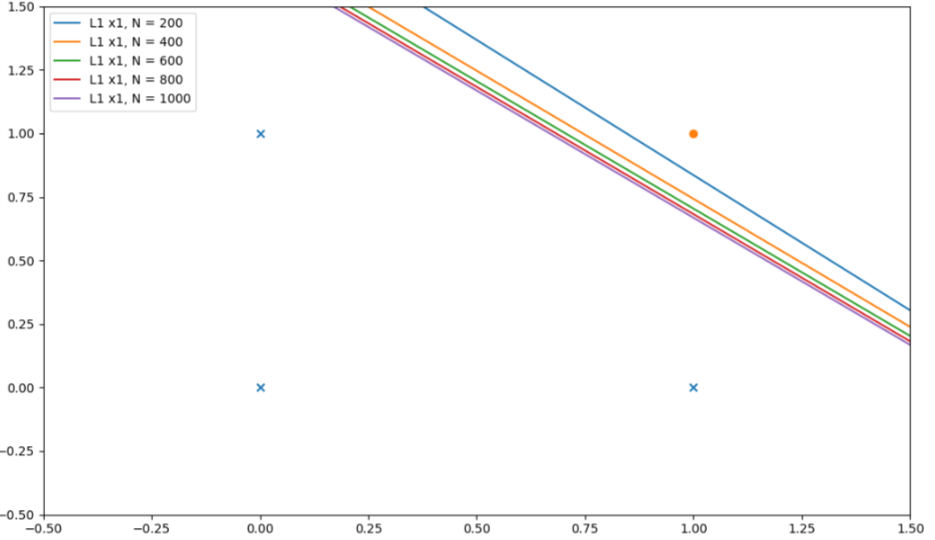


그림 4과 같이 Node x1의 그래프가 점점 내려가는 것을 볼 수 있습니다. 또, ReLU와 마찬가지로 그림 4과 같은 모양이 나타난다는 것을 알 수 있습니다.

* + 1. **OR Gate**
       1. **ReLU 사용**

Learning rate은 0.2, Training 횟수는 50번을 적용하였습니다. Seed는 ‘20181114’를 적용하였습니다. Hidden Layer의 수는 0이고 Input Layer의 노드 수는 2, Output Layer의 노드 수는 1로 적용하였습니다. OR에서도 ReLU를 적용할 경우 net이 음수가 되는 경우가 존재하고 그 결과 기울기가 0, f(net)값이 0이 되므로 학습이 안되는 경우가 많았습니다. 따라서 AND에서와 마찬가지로 ReLU보다는 Leaky ReLU를 사용하여 음수일때도 학습이 되도록 만들어야 합니다.

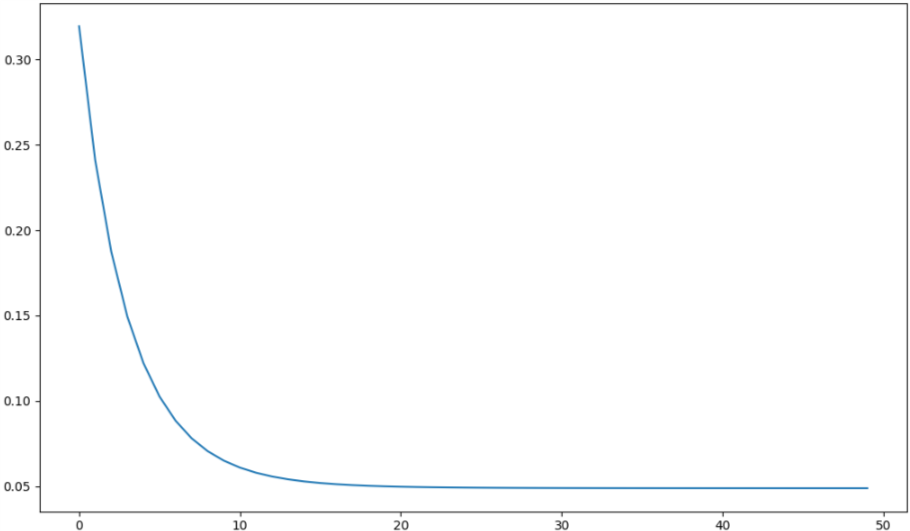


그림 5 : Error(OR, ReLU)

그림 5를 보면 Error가 점점 줄어드는 것을 알 수 있습니다.

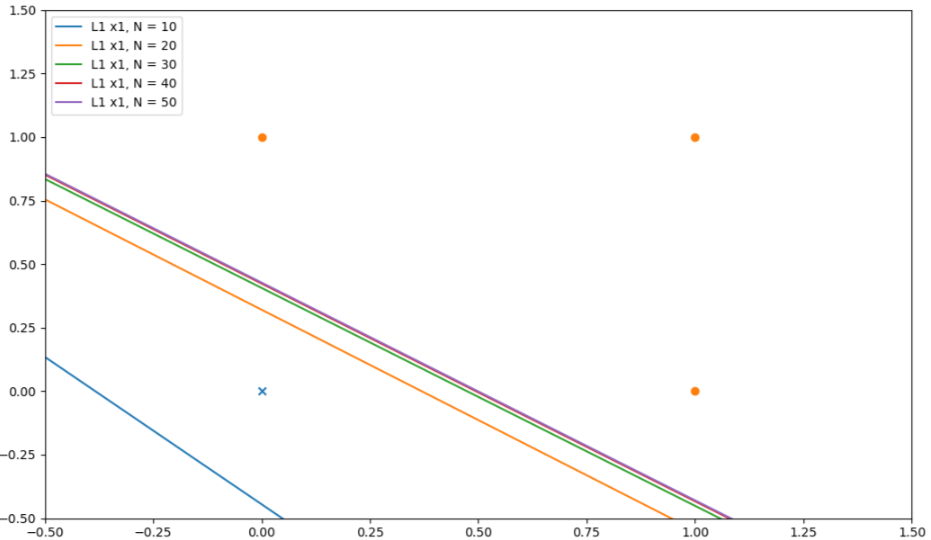


그림 6 : Linear Graph(OR, ReLU, Node(x1))

그림 6을 보면 그래프가 점점 올라가는 것을 알 수 있습니다. 그리고 Output Layer의 Node x1이 위와 같은 직선을 갖는다는 것을 알 수 있습니다.

* + - 1. **Sigmoid 사용**

Training 횟수는 600번을 적용하였고 나머지는 위의 ReLU와 같이 적용하였습니다. AND때와 마찬가지로 ReLU보다는 느리지만 net이 음수일때도 잘 학습시켜줍니다.

그림 7도 마찬가지로 Error가 점점 감소합니다.

그림 8 : Linear Graph(OR, Sigmoid, Node(x1))

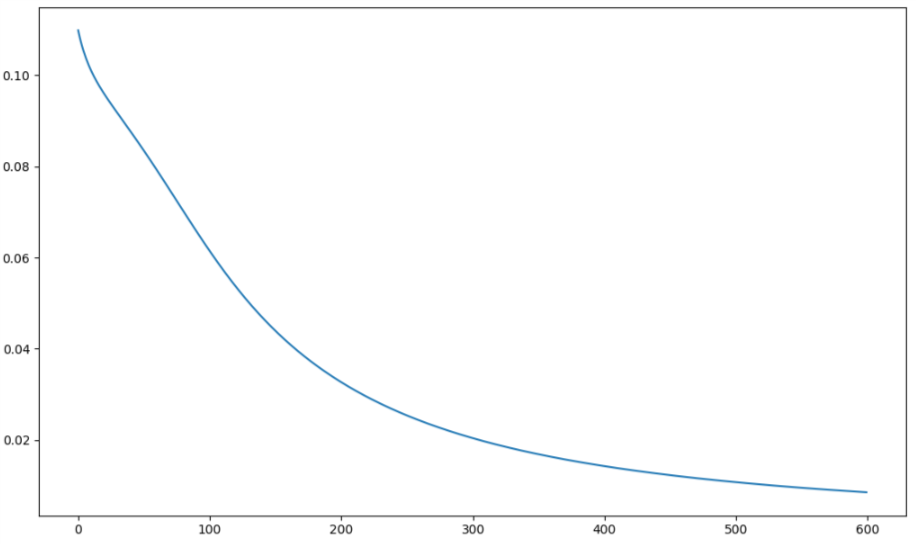
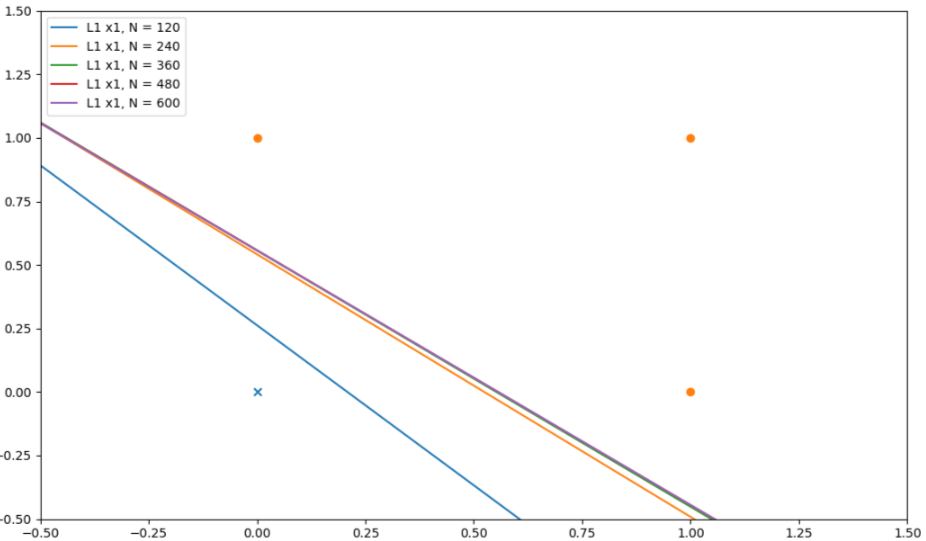


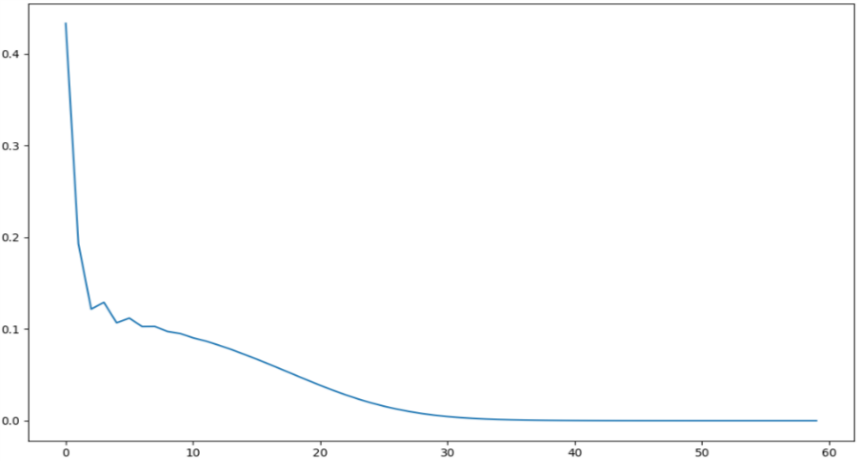
그림 7 : Error(OR, Sigmoid)

그림 8를 보면 ReLU를 적용했을 때와 비슷하게 직선이 점점 올라갑니다. 또한 이 직선이 Node x1의 그래프라는 것을 알 수 있습니다.

* + 1. **XOR Gate**
       1. **ReLU 사용**

Learning rate은 0.2, Training 횟수는 60번을 적용하였습니다. Seed는 ‘20181114’를 적용하였습니다. Hidden Layer의 수는 1이고 Input Layer의 노드 수는 2, Hidden 노드의 수는 2, Output Layer의 노드 수는 1로 적용하였습니다. XOR에서도 ReLU를 적용할 경우 net이 음수가 되는 경우가 존재하고 그 결과 기울기가 0, f(net)값이 0이 되므로 학습이 안되는 경우가 많았습니다. 따라서 위의 경우와 마찬가지로 ReLU보다는 Leaky ReLU를 사용하여 음수일때도 학습이 되도록 만들어야 합니다.

그림 9 : Error(XOR, ReLU)



이번에도 Error는 그림 9처럼 감소하는 것을 알 수 있었습니다. 이번에는 Error가 오히려 늘어나는 구간이 존재하였습니다. 하지만 그것도 잠깐이었고 다시 감소하는 경향을 보여주었습니다.

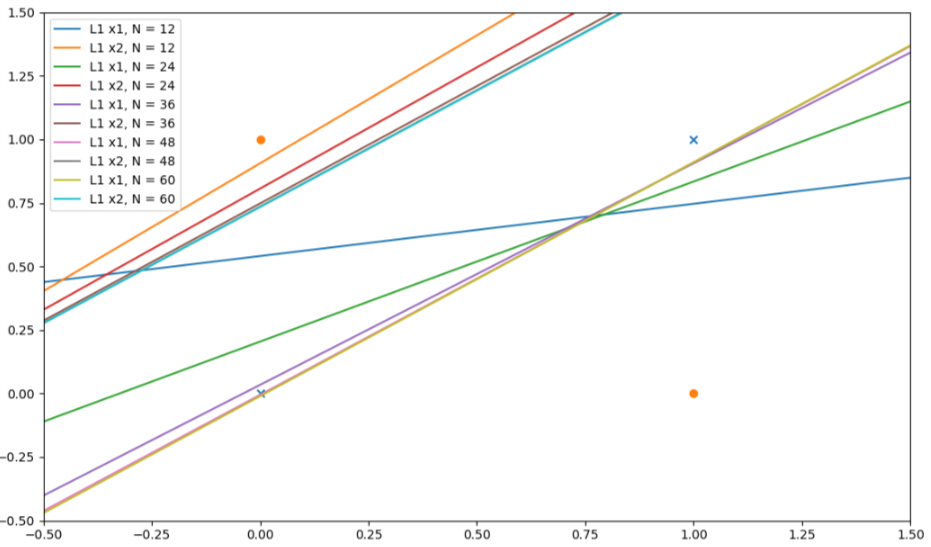


그림 10 : Linear Graph(XOR, ReLU, Node(L1 x1, L1 x2))

그림 10를 보면 선이 많아져서 알아보기 힘들지만 자세히 보면 알 수 있습니다. 이번에는 Hidden Layer 1의 x1, x2 노드들의 직선을 나타냈습니다. 왼쪽 위에 몰려 있는 직선들이 x2의 직선들, 오른쪽 아래에 몰려 있는 직선들이 x1의 직선들입니다. 각각 x2는 오른쪽 아래로, x1의 직선은 왼쪽 위로 올라가는 것을 알 수 있습니다. 또한 각각의 직선이 점들을 잘 나누어 주는 것을 알 수 있습니다.

* + - 1. **Sigmoid 사용**

Training 횟수를 8000번으로 적용하였고 나머지는 위의 ReLU와 같이 적용하였습니다. 이번에는 ReLU보다 훨씬 많은 횟수를 학습시켰습니다. Layer가 깊어질수록 1보다 작은 값이 계속 곱해지므로 Training속도가 점차 느려지는 것입니다. 따라서 속도를 올리기 위해서는 Adam과 같은 방법을 사용해야 합니다.

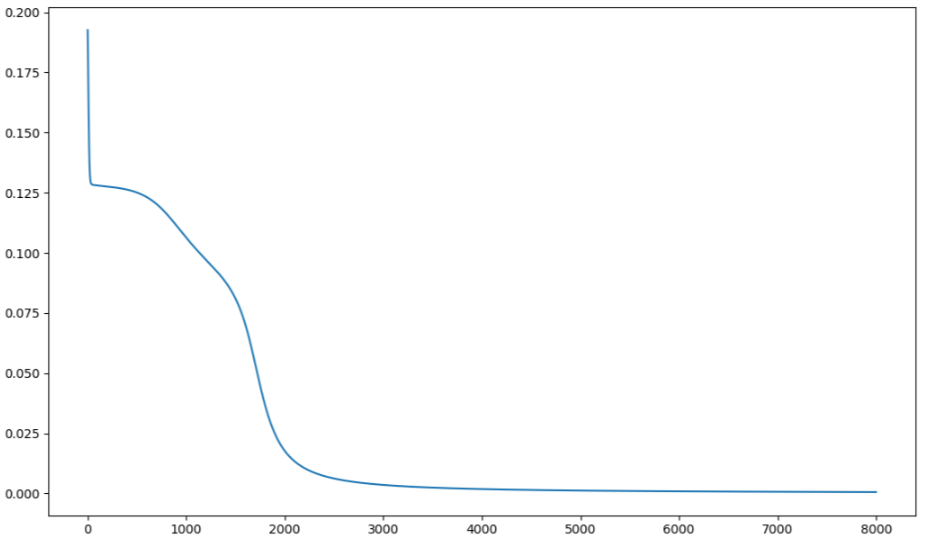


그림 11 : Error(XOR, Sigmoid)

이번에는 다소 원만하게 Error가 감소하였습니다.

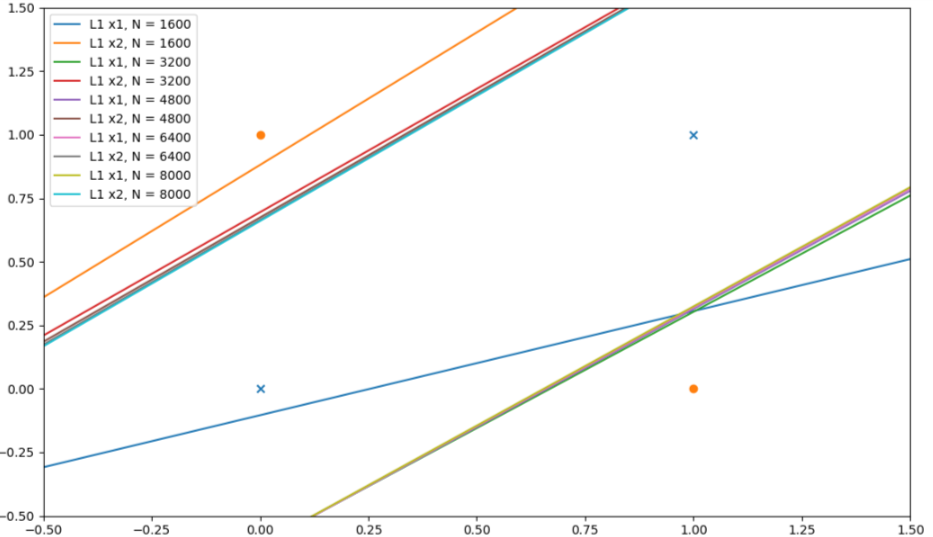


그림 12 : Linear Graph(XOR, Sigmoid, Node(L1 x1, L1 x2))

이번 그래프도 복잡하게 보이지만 ReLU와 거의 비슷하단 것을 할 수 있습니다. ReLU와 마찬가지로 왼쪽 위는 x2의 직선이고 오른쪽 아래는 x1의 직선입니다. 각각 x2는 오른쪽 아래로 내려가고, x1은 왼쪽 위로 올라가는 것을 알 수 있습니다. 그리고 이 방법 역시 두 직선으로 점들을 잘 구분해 준다는 것을 알 수 있습니다.

* 1. 도넛 구분 실험
     1. **Leaky ReLU 사용**

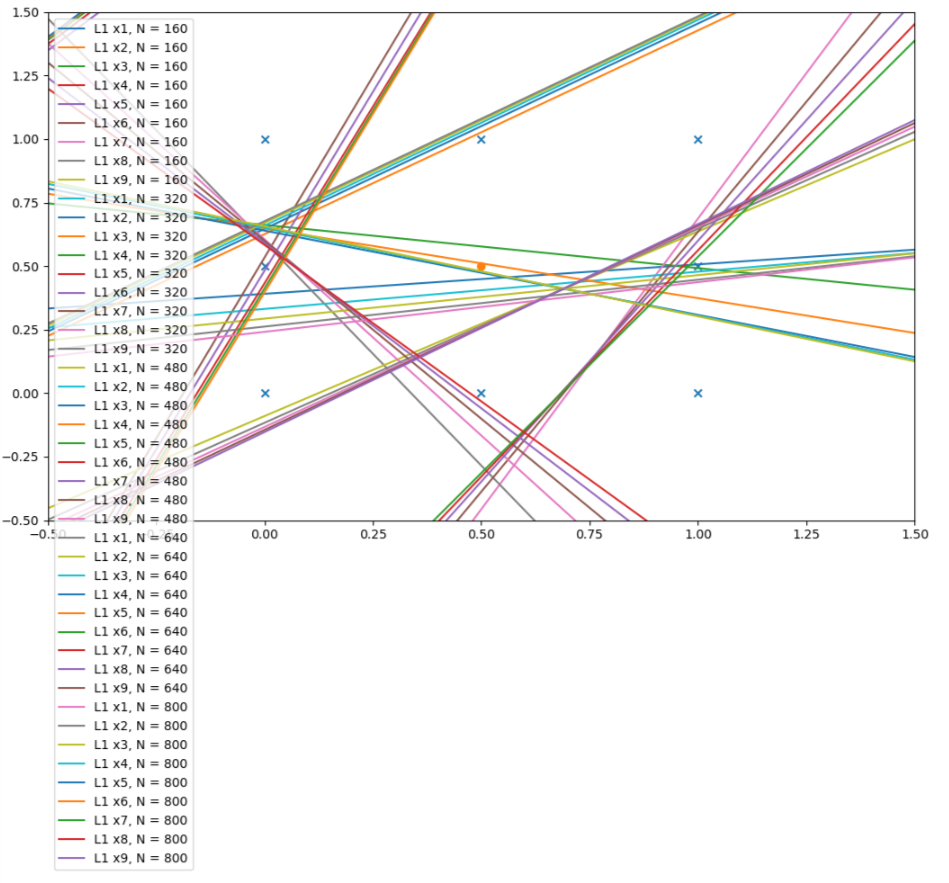
ReLU를 사용할 경우 Node가 많아서 net값이 음수가 나올 가능성이 높습니다. 거의 대부분의 경우 계산 과정에서 net값이 음수가 나와 학습이 안되었습니다. 따라서 ReLU를 수정하여 Leaky ReLU를 사용하였습니다. net값이 음수일때는 0.1\*x의 값을 return하였고 미분한 함수는 0.1을 return하였습니다. Hidden Layer의 Layer 수는 1개, node의 수는 9개를 사용했고 seed값은 ‘20181114’를 사용하였습니다. Training은 800번 진행하였습니다.



그림 13 : Error(도넛, Leaky ReLU)

Error는 0~100 구간에서 격렬하게 진동하면서 내려갔습니다.

그림 14 : Linear Graph(도넛, Leaky ReLU, L1 Node(x1~x9))



hidden layer의 노드 수가 늘어나서 그래프를 분석하기가 매우 어려워졌습니다. 하지만 Layer를 1개만 넣었기 때문에 그나마 분석하기 수월합니다. 각각의 직선뭉치는 가운데 점과 바깥 점들을 구분시켜 주고 있습니다. 9개의 직선이지만 그래프에서는 8개의 직선 뭉치만 보입니다. 한 뭉치는 바깥 구간에 있어서 보이지 않는 상태입니다.

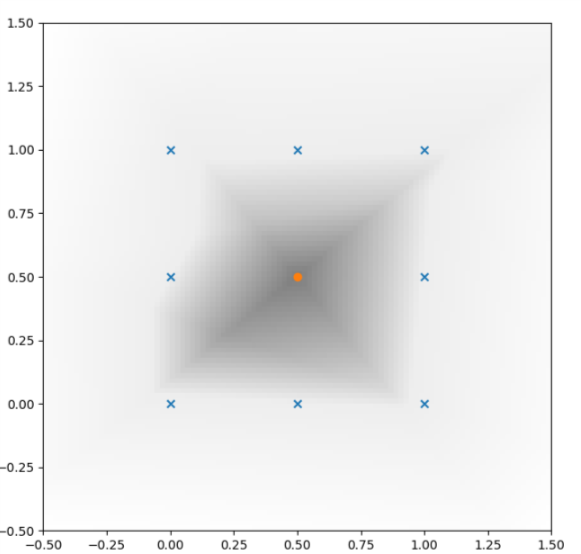


그림 15 : 도넛 모양 Graph(도넛, Leaky ReLU)

의도했던 모양과는 조금 다르지만 가운데를 기준으로 까맣게 나오는 것을 알 수 있습니다. 요즘엔 네모 모양의 도넛도 있으니까 이것도 도넛 모양이라고 할 수 있다고 생각합니다. 9개의 직선들이 바깥점과 안의 점을 구분 시켜주므로 가운데만 까맣게 나옵니다. 하지만 직선이 적고, Leaky ReLU를 사용했으므로 동그란 모양이 아닌 네모난 모양이 나왔습니다. 뒤의 추가 실험으로 노드 수를 늘려서 더 동그란 모양에 가깝게 만들어볼 계획입니다.

* + 1. **Sigmoid 사용**

ReLU와는 다르게 net값이 음수일때 0이 나오지 않아서 얼마든지 사용이 가능합니다. 하지만 다른 실험과도 마찬가지로 ReLU보다는 Training 횟수가 더 많이 필요했습니다. 따라서 이번에는 Training 횟수를 6000번으로 설정하고 나머지는 위 실험과 동일하게 진행하였습니다. ReLU보다는 값의 범위는 좁지만 더 세밀한 값이 나오므로 더욱 동그랗게 나올 것입니다.

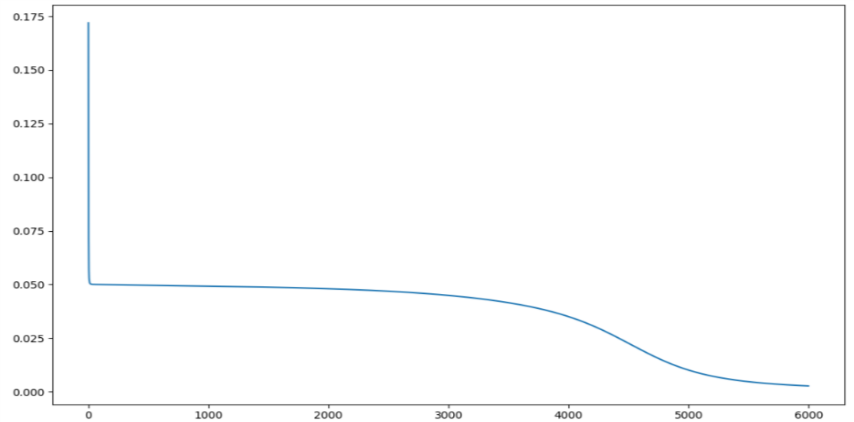
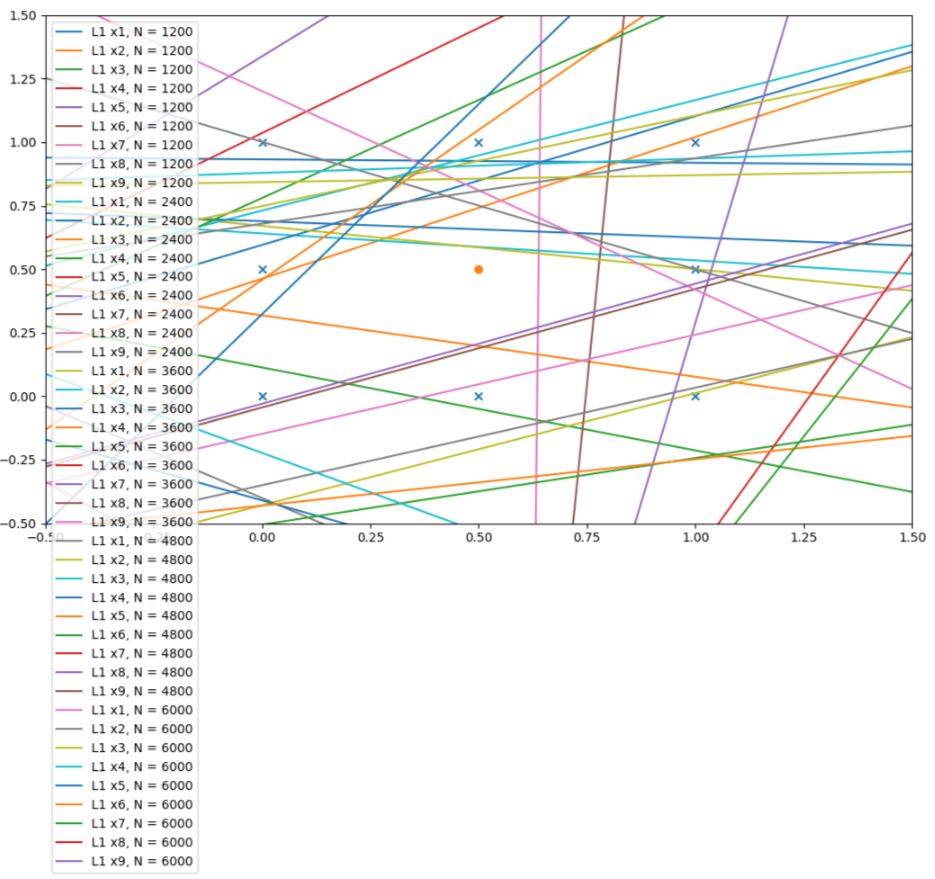


그림 16 : Error(도넛, Sigmoid)

위의 실험과는 다르게 원만하게 감소하는 형태를 가집니다. 3000번까지 감소량이 거의 없다가 4000번부터 빠르게 내려갔습니다.

그림 17 : Linear Graph(도넛, Sigmoid, L1 Node(x1~x9))



이번 그래프는 정말 복잡하기 때문에 거의 분석이 불가능합니다. 하지만 직선들이 가운데만 비워 두고 모두 바깥 점과 안의 점을 구분 시켜주고 있습니다.

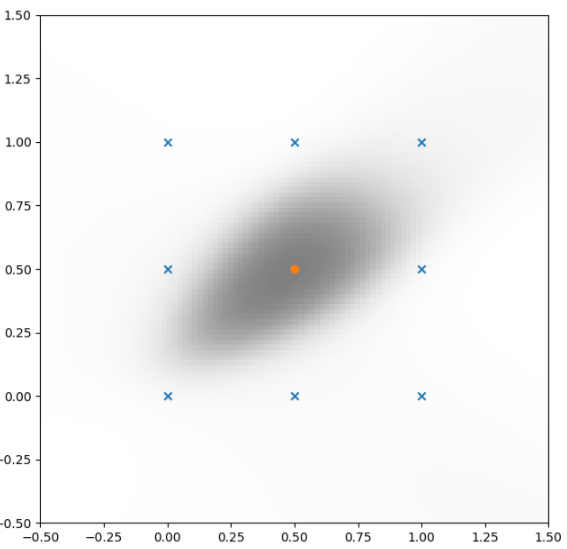


그림 18 : 도넛 모양 Graph(도넛, Sigmoid)

Sigmoid는 거의 원에 가깝게 보여주고 있습니다. 노드 수를 충분히 늘리면 더욱 원에 가까워질 것이라고 생각합니다.

1. **결론**

Seed, training 횟수, Activation Function, Hidden Layer의 수, 각 Layer의 node수 등 정말 다양한 요인들이 작용하여 Training이 진행됩니다. 이처럼 많은 요인들이 결과에 영향을 미치므로 어떤 문제에 대하여 어떤 요인을 어떻게 설정할지는 경험적인 센스가 필요할 것 같습니다.

1. **추가 실험**

Hidden node의 개수를 조절하면 도넛 실험에서 그래프 모양이 점점 원에 가까워진다는 것을 알 수 있었습니다. 하지만 레이어 수를 늘리면 오히려 그래프가 점과 점 사이를 비집고 나가는 현상을 보였습니다. 그래서 레이어의 수는 1로 하고 노드의 수만 조절하기로 하였습니다. Hidden 노드의 수를 50개, 100개로 늘려가면서 그래프를 그려보았습니다. Seed는 ‘20181114’입니다. 그림 19는 Hidden node의 개수가 50개이고 사용한 함수는 sigmoid이며 10000번 학습시켰습니다. 그림 20은 Hidden node의 개수가 100개이고 나머지는 위와 같습니다. 모양이 점점 원형에 가까워지는 것을 볼 수 있었습니다. 이 추가 실험은 시간이 오래 걸리기 때문에 위의 실험들과는 다르게 성능이 좋은 데스크탑에서 진행하였습니다.

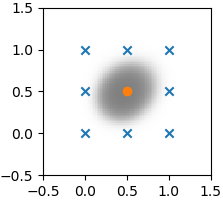


그림 20 : 도넛 모양 Graph(도넛, Sigmoid, 500개)

그림 19 : 도넛 모양 Graph(도넛, Sigmoid, 50개)

