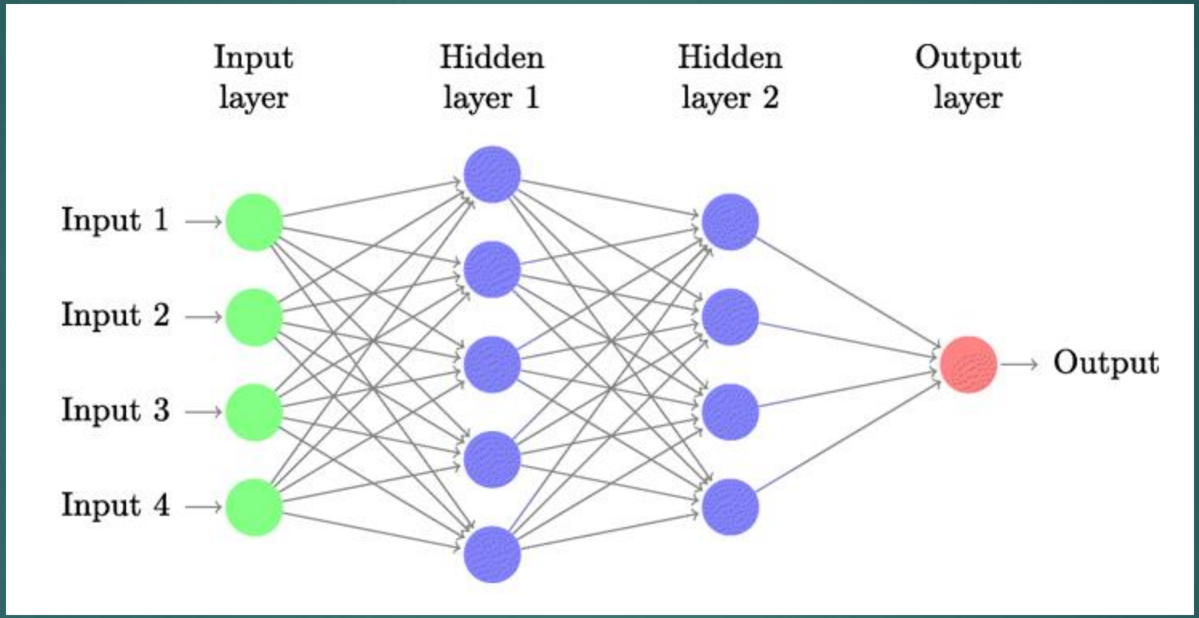
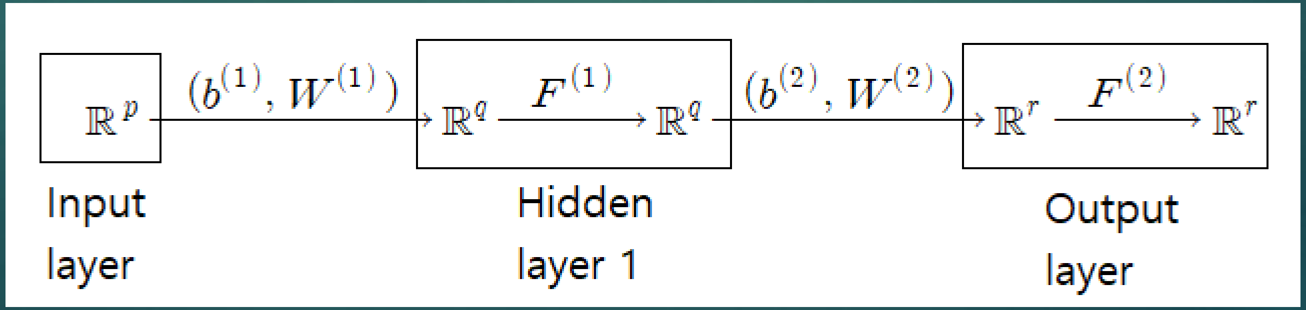
인공지능프로그래밍

게임콘텐츠학과 박경수

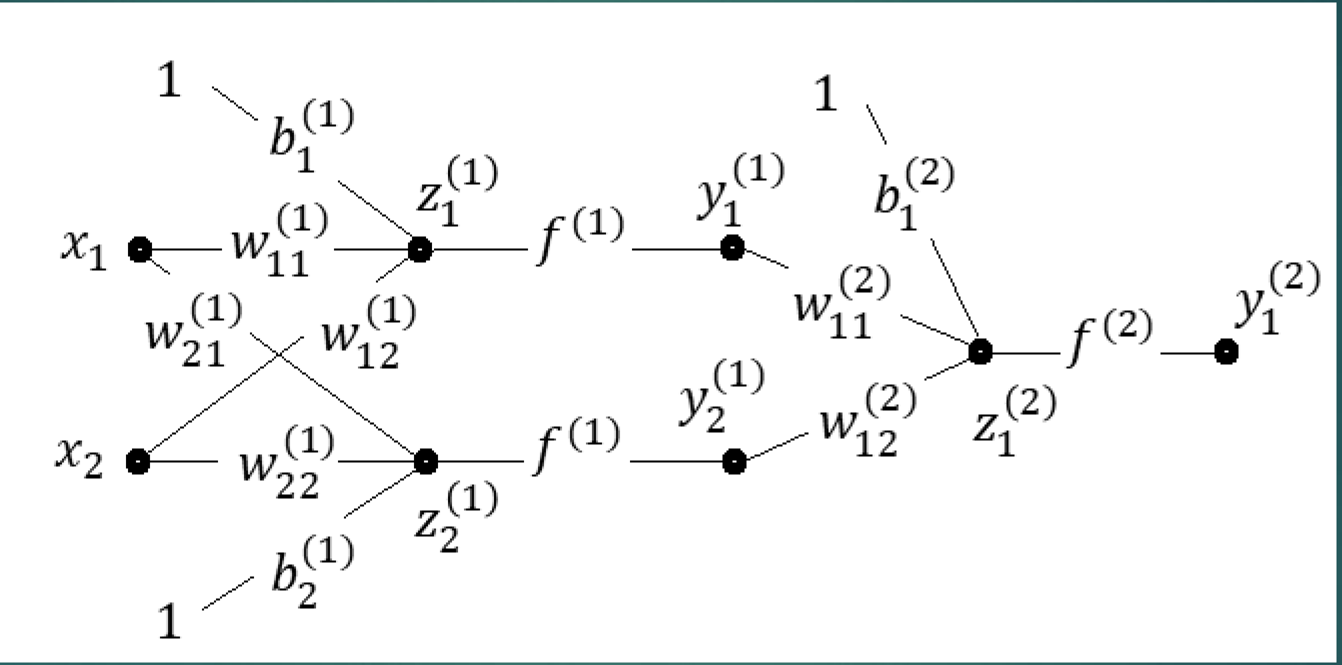
https://github.com/ggorr/Machine-Learning/tree/master/Python

Feedforward Network



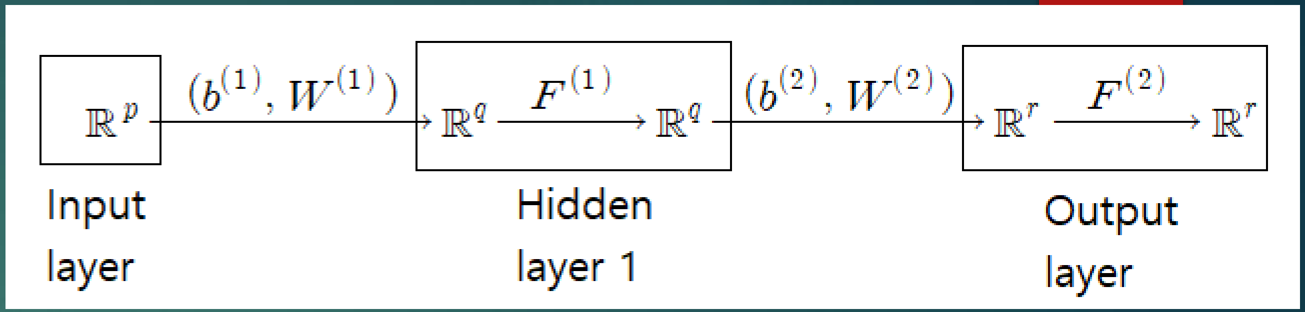


Example



1
$b_1^{(1)}$
$z_1^{(1)}$
$y_1^{(1)}$
1
$b_1^{(2)}$
x_1
$w_{11}^{(1)}$
$f^{(1)}$
$w_{11}^{(2)}$
$w_{21}^{(1)}$
$w_{12}^{(1)}$
$f^{(2)}$
$y_1^{(2)}$
$y_2^{(1)}$
$w_{12}^{(2)}$
$z_1^{(2)}$
x_2
$w_{22}^{(1)}$
$f^{(1)}$
$z_2^{(1)}$
$b_2^{(1)}$
1

Notation



- Input data

$$X = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{p1} & \cdots & x_{pn} \end{bmatrix}, \ X_i = \begin{bmatrix} x_{1i} \\ \vdots \\ x_{pi} \end{bmatrix}$$

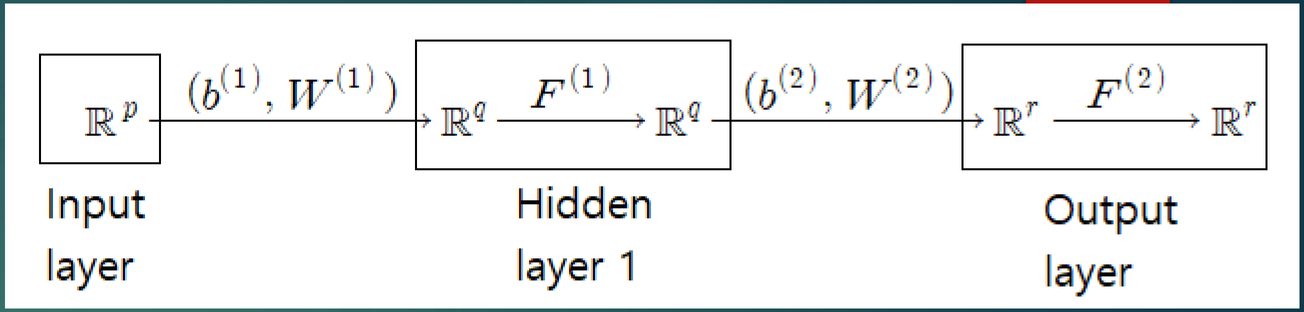
- Bias and weight

$$b^{(1)} = \begin{bmatrix} b_1^{(1)} \\ \vdots \\ b_p^{(1)} \end{bmatrix}, W^{(1)} = \begin{bmatrix} w_{11}^{(1)} & \cdots & w_{1p}^{(1)} \\ \vdots & \ddots & \vdots \\ w_{q1}^{(1)} & \cdots & w_{qp}^{(1)} \end{bmatrix}$$

- Activation

$$F^{(1)} = \begin{bmatrix} f^{(1)} \\ \vdots \\ f^{(1)} \end{bmatrix}$$ where $f^{(1)}(x) = \sigma(x) = \frac{1}{1+e^{-x}}$, the sigmoid function

- Hidden layer



$$Z_i^{(1)} = \begin{bmatrix} z_{1i}^{(1)} \\ \vdots \\ z_{qi}^{(1)} \end{bmatrix} = b^{(1)} + W^{(1)} X_i,$$

i.e. $Z^{(1)} = b^{(1)} + W^{(1)} X$

$$Y_i^{(1)} = \begin{bmatrix} y_{1i}^{(1)} \\ \vdots \\ y_{qi}^{(1)} \end{bmatrix} = F^{(1)}\left(Z_i^{(1)}\right) = \begin{bmatrix} f^{(1)}\left(z_{1i}^{(1)}\right) \\ \vdots \\ f^{(1)}\left(z_{qi}^{(1)}\right) \end{bmatrix},$$

i.e. $Y^{(1)} = F^{(1)}\left(Z^{(1)}\right)$

Loss

- Input data

  $$X_1, \dots, X_n \text{ with } X_i = (x_{1i}, \dots, x_{pi})$$

  - Matrix form

    $$X = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{p1} & \cdots & x_{pn} \end{bmatrix}$$

- Target

  $$T_1, \dots, T_n \text{ with } T_i = (t_{1i}, \dots, t_{ri})$$

  - Matrix from

    $$T = \begin{bmatrix} t_{11} & \cdots & t_{1n} \\ \vdots & \ddots & \vdots \\ t_{r1} & \cdots & t_{rn} \end{bmatrix}$$

- Output

$$Y_1^{(2)}, \dots, Y_n^{(2)} \text{ with } Y_i^{(2)} = \left(y_{1i}^{(2)}, \dots, y_{ri}^{(2)}\right)$$

  - Matrix from

$$Y = Y^{(2)} = \begin{bmatrix} y_{11}^{(2)} & \cdots & y_{1n}^{(2)} \\ \vdots & \ddots & \vdots \\ y_{r1}^{(2)} & \cdots & y_{rn}^{(2)} \end{bmatrix}$$

- MSE(Mean squared error)

$$E = \sum_{i=1}^{n} \frac{1}{2} \parallel Y_i^{(2)} - T_i \parallel^2$$

Goal

- Find a feedforward network which minimizes the loss

$$E = \sum_{i=1}^{n} \frac{1}{2} \| Y_i^{(2)} - T_i \|^2$$
$$= \sum_{i=1}^{n} \frac{1}{2} \sum_{j=1}^{r} \left(y_{ji}^{(2)} - t_{ji} \right)^2$$

- Approach

  Find $b^{(j)}$ and $W^{(j)}$ using stochastic gradient descent

Computation

$$\mathbb{R}^p \xrightarrow{(b^{(1)}, W^{(1)})} \mathbb{R}^q \xrightarrow{F^{(1)}} \mathbb{R}^q \xrightarrow{(b^{(2)}, W^{(2)})} \mathbb{R}^r \xrightarrow{F^{(2)}} \mathbb{R}^r$$

$$X \xrightarrow{(b^{(1)}, W^{(1)})} Z^{(1)} \xrightarrow{F^{(1)}} Y^{(1)} \xrightarrow{(b^{(2)}, W^{(2)})} Z^{(2)} \xrightarrow{F^{(2)}} Y^{(2)}$$

- Goal: to compute gradient of loss with respect to $W^{(i)}$ and $b^{(i)}$
- Loss

$$\begin{aligned} E &= \sum_{i=1}^{n} \frac{1}{2} \parallel Y_i^{(2)} - T_i \parallel^2 \\ &= \sum_{i=1}^{n} \frac{1}{2} \sum_{j=1}^{r} \left(y_{ji} - t_{ji}\right)^2 \end{aligned}$$

- Gradients

$$\nabla_{Y^{(2)}} E = \begin{bmatrix} \frac{\partial E}{\partial y_{11}} & \cdots & \frac{\partial E}{\partial y_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial y_{r1}} & \cdots & \frac{\partial E}{\partial y_{rn}} \end{bmatrix} = \begin{bmatrix} y_{11} - t_{11} & \cdots & y_{1n} - t_{1n} \\ \vdots & \ddots & \vdots \\ y_{r1} - t_{r1} & \cdots & y_{rn} - t_{rn} \end{bmatrix} = Y^{(2)} - T$$

$$\mathbb{R}^p \xrightarrow{(b^{(1)}, W^{(1)})} \mathbb{R}^q \xrightarrow{F^{(1)}} \mathbb{R}^q \xrightarrow{(b^{(2)}, W^{(2)})} \mathbb{R}^r \xrightarrow{F^{(2)}} \mathbb{R}^r$$

$$X \xrightarrow{(b^{(1)}, W^{(1)})} Z^{(1)} \xrightarrow{F^{(1)}} Y^{(1)} \xrightarrow{(b^{(2)}, W^{(2)})} Z^{(2)} \xrightarrow{F^{(2)}} Y^{(2)}$$

$$\nabla_{Z^{(2)}} E = \begin{bmatrix} f^{(2)\prime}(z_{11}) \frac{\partial E}{\partial y_{11}} & \cdots & f^{(2)\prime}(z_{1n}) \frac{\partial E}{\partial y_{1n}} \\ \vdots & \ddots & \vdots \\ f^{(2)\prime}(z_{r1}) \frac{\partial E}{\partial y_{r1}} & \cdots & f^{(2)\prime}(z_{rn}) \frac{\partial E}{\partial y_{rn}} \end{bmatrix}$$

$$= JF^{(2)}\left(Z^{(2)}\right) * \nabla_{Y^{(2)}} E$$

$$\nabla_{W^{(2)}} E = \nabla_{Z^{(2)}} E * Y^{(1)T}$$

$$\nabla_{b^{(2)}} E = \begin{bmatrix} \sum_{i=1}^{n} \frac{\partial E}{\partial z_{1i}^{(2)}} \\ \vdots \\ \sum_{i=1}^{n} \frac{\partial E}{\partial z_{ri}^{(2)}} \end{bmatrix}$$

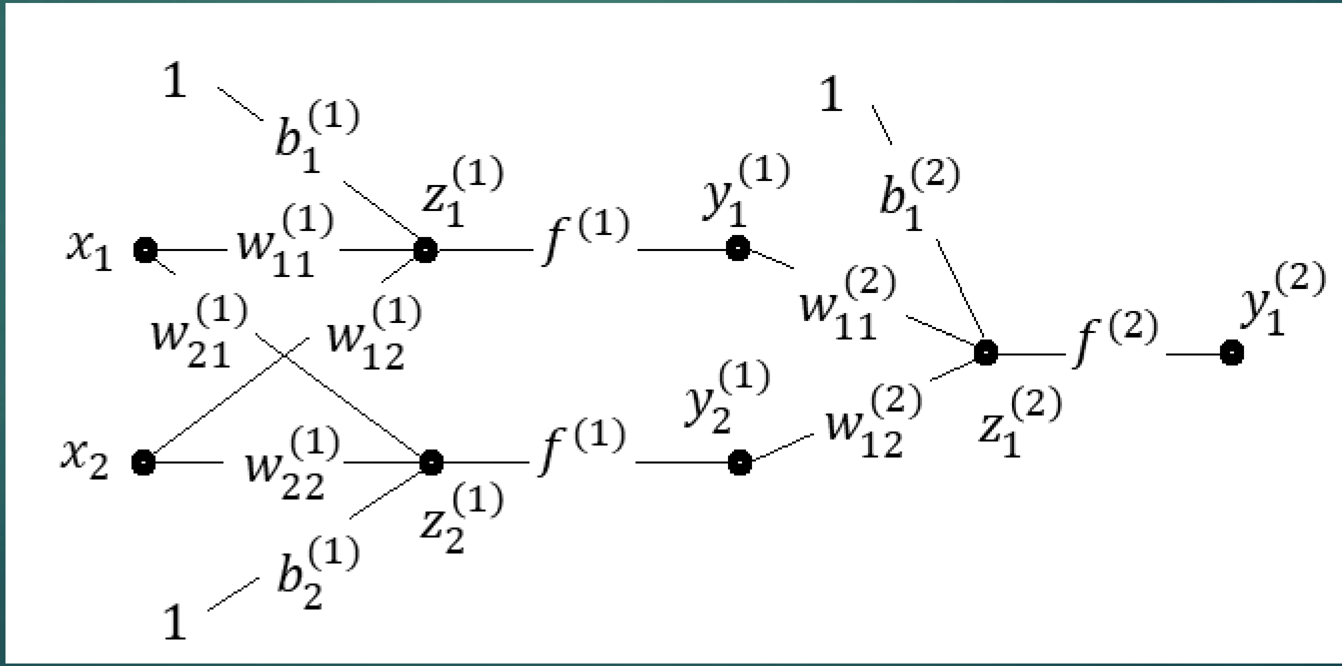
$$\nabla_{Y^{(1)}} E = W^{(2)T} \nabla_{Z^{(2)}} E$$

where $*$ is the Hadamard product

Example: XOR

- Input: $\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$
  Target: $\begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}$

Network:



```python
import numpy as np

X = np.array([[0., 0., 1., 1.], [0., 1., 0., 1.]])
target = np.array([[0., 1., 1., 0.]])

# input layer
# number of nodes
p0 = 2
# number of samples
n = 4

# hidden layer 1
# number of nodes
p1 = 2
# bias
b1 = np.random.rand(p1, 1)
# weight
W1 = np.random.rand(p1, p0)
# activation
f1 = lambda x: 1 / (1 + np.exp(-x))

# output layer
# number of nodes
p2 = 1
# bias
b2 = np.random.rand(p2, 1)
# weight
W2 = np.random.rand(p2, p1)
# activation
f2 = lambda x: 1 / (1 + np.exp(-x))
```

```python
lr = 0.1
epoch = 0
while True:
    # forward propagation
    z1 = b1 + np.matmul(W1, X)
    y1 = f1(z1)
    z2 = b2 + np.matmul(W2, y1)
    y2 = f2(z2)
    # error
    error = .5 * np.sum((target - y2) ** 2)
    if error < 0.0001 or epoch >= 10000:
        break

    # backward propagation
    grad_y2 = y2 - target
    grad_z2 = y2 * (1 - y2) * grad_y2
    grad_b2 = np.sum(grad_z2, axis=1, keepdims=True)
    grad_W2 = np.matmul(grad_z2, y1.T)
    grad_y1 = np.matmul(W2.T, grad_z2)
    grad_z1 = y1 * (1 - y1) * grad_y1
    grad_b1 = np.sum(grad_z1, axis=1, keepdims=True)
    grad_W1 = np.matmul(grad_z1, X.T)

    # update
    b1 -= lr * grad_b1
    W1 -= lr * grad_W1
    b2 -= lr * grad_b2
    W2 -= lr * grad_W2
    epoch += 1

print(y2)
```

Using class

```python
import numpy as np
import matplotlib.pyplot as plt

sigmoid = lambda x: 1 / (1 + np.exp(-x))
dsigmoid = lambda y: y * (1 - y)
tanh = np.tanh
dtanh = lambda y: 1 - y * y
relu = lambda x: (x > 0) * x
drelu = lambda y: (y > 0).astype(float)

class Layer:
    def __init__(self, xdim, ydim, f, df):
        """
        :param xdim: input dimension
        :param ydim: output dimension
        :param f: activation function
        :param df: Jacobian of f
        """
        self.xdim = xdim
        self.ydim = ydim
        self.f = f
        self.df = df
        # bias
        self.B = np.random.rand(ydim, 1)
        # weight
        self.W = np.random.rand(ydim, xdim)
        # input
        self.x = None
        # output
        self.y = None
        # gradient of bias
        self.gradB = np.empty_like(self.B)
        # gradient of weight
        self.gradW = np.empty_like(self.W)
```

```python
    def forProp(self, x):
        """forward propagation"""
        self.x = x
        self.y = self.f(self.B + np.matmul(self.W, x))
        return self.y

    def backProp(self, gradY):
        """
        :param gradY: gradient of loss w.r.t. y
        :return: gradient of loss w.r.t. x
        """
        gradZ = self.df(self.y) * gradY
        self.gradB = np.sum(gradZ, axis=1, keepdims=True)
        self.gradW = np.matmul(gradZ, self.x.T)
        return np.matmul(self.W.T, gradZ)

    def update(self, lr):
        self.B -= lr * self.gradB
        self.W -= lr * self.gradW


X = np.array([[0., 0., 1., 1.], [0., 1., 0., 1.]])
target = np.array([[0., 1., 1., 0.]])
# layers = [Layer(2, 5, sigmoid, dsigmoid),
#           Layer(5, 1, sigmoid, dsigmoid)]
layers = [Layer(2, 2, sigmoid, dsigmoid),
          Layer(2, 1, sigmoid, dsigmoid)]
```

```python
lr = 0.1
epoch = 0
while True:
    # forward propagation
    y = X
    for lay in layers:
        y = lay.forProp(y)

    # error
    error = .5 * np.sum((target - y) ** 2)
    if error < 0.0001 or epoch >= 10000:
        break

    # backward propagation
    grad = y - target
    for lay in layers[::-1]:
        grad = lay.backProp(grad)

    # update
    for lay in layers:
        lay.update(lr)
    epoch += 1

print(y)
```

```python
X1 = np.linspace(-1, 2, 61)
Y1 = np.linspace(-1, 2, 61)
X1, Y1 = np.meshgrid(X1, Y1)
Z1 = np.empty_like(X1)
for x1, y1, z1 in zip(X1, Y1, Z1):
    xy = np.vstack((x1, y1))
    for lay in layers:
        xy = lay.forProp(xy)
    z1[:] = xy
cs = plt.contour(X1, Y1, Z1)
plt.clabel(cs)
plt.plot([0, 1, 1, 0, 0], [0, 0, 1, 1, 0])
plt.show()
```

Activation functions

- Sigmoid

  $\sigma(x) = \frac{1}{1+e^{-x}}, \sigma'(x) = y(1-y)$

- ReLU

  $f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}, f'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$

- Hyper tangent

  $\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tanh' x = 1 - y^2$

- Softmax

  $\mathrm{F}(x_1, \dots, x_n) = \left(\frac{e^{x_1}}{e^{x_1} + \dots + e^{x_n}}, \dots, \frac{e^{x_n}}{e^{x_1} + \dots + e^{x_n}}\right)$

- Softplus

  $f(x) = \log(1 + e^x)$

Loss Functions

- MSE(Mean squared error)

  $$E = \sum_{i=1}^{n} \frac{1}{2} \parallel Y_i^{(2)} - T_i \parallel^2$$

- Binary crossentropy

  $$E = -\sum_{i=1}^{n} \left(t_i \log y_i^{(2)} + (1 - t_i) \log \left(1 - y_i^{(2)} \right) \right)$$

  - Output dimension = 1, i.e. $r = 1$
  - Activation function = sigmoid

- Categorical crossentropy

  $$E = -\sum_{i=1}^{n} \sum_{j=1}^{r} t_{ji} \log y_{ji}^{(2)}$$

  - Output dimension > 1, i.e. $r > 1$
  - Activation function = softmax