Definition

K: number of layers, $k \in \{1, \cdots, K\}$

$N_k - 1$: number of nodes in k th layer

N: number of observations

$A^{(k-1)}: \quad N \times N_{k-1}$ matrix, **input** of k th layer

$$A^{(0)} = X^T, N \times N_0$$
$$A^{(k)}, \quad N \times N_k$$
$$A^{(k)} = \begin{bmatrix} \text{ones}(N,1) & a^{(k)} \end{bmatrix} = \begin{bmatrix} A_1^{(k)} & A_2^{(k)} \cdots A_{N_k}^{(k)} \end{bmatrix}, \quad A_1^{(k)} = ones(N,1)$$

$W^{(k)}: \quad N_{k-1} \times (N_k - 1)$matrix, weight in layer k

In layer k: $A_j^{(k-1)} \to A_i^{(k)}$, weight is $W_{ji}^{(k)}$

$$W^{(k)} = \begin{bmatrix} W_1^{(k)} & W_2^{(k)} \cdots W_i^{(k)} \cdots W_{N_k-1}^{(k)} \end{bmatrix}$$

$$A^{(k-1)} W_i^{(k)} = \sum_{j=1}^{N_{k-1}} A_j^{(k-1)} W_{ji}^{(k)} = Z_i^{(k)}, \quad i \in \{1, \cdots, N_k - 1\}$$

$Z^{(k)}: \quad N \times (N_k - 1)$ matrix, Intermediate variables

$$Z^{(k)} = A^{(k-1)} W^{(k)}$$
$$Z^{(k)} = \begin{bmatrix} Z_1^{(k)} & Z_2^{(k)} \cdots Z_i^{(k)} \cdots Z_{N_k-1}^{(k)} \end{bmatrix}$$
$$\frac{\partial J}{\partial W^{(k)}} = \left(A^{(k-1)}\right)^T \frac{\partial J}{\partial Z^{(k)}}$$
$$\frac{\partial J}{\partial A^{(k-1)}} = \frac{\partial J}{\partial Z^{(k)}} \left(W^{(k)}\right)^T$$

$\phi(\cdot)$: activation function **sigmoid function** for layer $1 \to K - 1$

$$\frac{d\phi(\cdot)}{d(\cdot)} = \phi(\cdot) \odot (1. - \phi(\cdot))$$

In last layer K, activation function could be $softmax(\cdot)$, **not** $\phi(\cdot)$

$a^{(k)}: \quad N \times N_k - 1$ matrix, **output** of k th layer

$$
\begin{aligned}
a^{(k)} &= \phi(Z^{(k)}) \\
\frac{\partial J}{\partial Z^{(k)}} &= \frac{\partial J}{\partial a^{(k)}} \odot \left[a^{(k)} \odot \left(1. - a^{(k)}\right)\right]
\end{aligned}
$$

$M^{(k)}$: $N \times N_k$matrix, connection between **input** of $k+1$ th layer && **output** of k th layer

$$
\begin{aligned}
A^{(k)} &= [\mathrm{ones}(N,1), a^{(k)}] = a^{(k)}[\mathrm{zeros}(N_k-1,1), I_{N_k-1}] + \mathrm{ones}(N,1)\cdot[1,0,\cdots,0] \\
&= a^{(k)}M^{(k)} + \mathrm{ones}(N,1)\cdot[1,0,\cdots,0] \\
\frac{\partial J}{\partial a^{(k)}} &= \frac{\partial J}{\partial A^{(k)}}[\mathrm{zeros}(N_k-1,1), I_{N_k-1}]^T = \frac{\partial J}{\partial A^{(k)}}\left(M^{(k)}\right)^T
\end{aligned}
$$

J: Cost function of network

- When size of $a^{(K)}$ is $N \times N_K - 1 = N \times 1$, activation function in layer K: $a^{(K)} = \phi(Z^{(K)})$

$$
J \equiv \sum\left(-Y^T \odot \log(a^{(K)}) - (1. - Y^T) \odot \log(1. - a^{(K)})\right)
$$

```
    a2 = sigmoid(z2);
    Loss = - y'.* log(a2) - (1. - y').* log(1. - a2);
    Cost = sum(Loss);
    % in function TwoLayerPerceptron.m
```

- When size of $a^{(K)}$ is $N \times N_K - 1 \neq N \times 1$, activation function in layer K: $a^{(K)} = \mathrm{softmax}(Z^{(K)})$

$$
J \equiv -\mathrm{tr}\left(Y \cdot \log(a^{(K)})\right)
$$

```
    a2 = softmax(z2);
    Cost = - trace( y * log(a2) );
    % in function TwoLayerPerceptron_softmax.m
```

After all, in those 2 cases:

$$
\frac{\partial J}{\partial Z^{(K)}} = \left(a^{(K)} - Y^T\right)
$$

```
    dJ_dz2 = (a2 - y');
    % in function TwoLayerPerceptron.m &&
 TwoLayerPerceptron_softmax.m
```

You can find details of proof here:
https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/

https://math.stackexchange.com/questions/945871/derivative-of-softmax-loss-function

Rule of back propagation

It is easy to verify that, when using **Cross Entropy** with $\text{softmax}(\cdot)$

$$\begin{aligned} J &= -Y \log\left(\frac{\exp(Z^{(K)})}{1^T \exp(Z^{(K)})}\right) \\ &= -Y \log\left(\exp(Z^{(K)})\right) + Y \cdot 1 \cdot \log(1^T \exp(Z^{(K)})) \\ &= -Y \cdot Z^{(K)} + \log(1^T \exp(Z^{(K)})) \end{aligned}$$

here 1 indicates Vector filled with 1, here we notice that if we sum up Y, then result is 1

$$\frac{\partial[-Y \cdot Z^{(K)}]}{\partial Z^{(K)}} = -Y^T$$

then

$$\frac{\partial \log(1^T \exp(Z^{(K)}))}{\partial Z^{(K)}} = \frac{1}{1^T \exp(Z^{(K)})} \odot \exp(a^{(K)}) = \frac{\exp(Z^{(K)})}{1^T \exp(Z^{(K)})}$$

finally

$$\frac{\partial J}{\partial Z^{(K)}} = \frac{\exp(Z^{(K)})}{1^T \exp(Z^{(K)})} - Y^T = \left(a^{(K)} - Y^T\right)$$

In **last** layer K

$$\frac{\partial J}{\partial Z^{(K)}} = \left(a^{(K)} - Y^T\right)$$

Connection between layer $k+1$ && layer k

$$\begin{aligned} \frac{\partial J}{\partial A^{(k)}} &= \frac{\partial J}{\partial Z^{(k+1)}} \left(W^{(k+1)}\right)^T \\ \frac{\partial J}{\partial a^{(k)}} &= \frac{\partial J}{\partial A^{(k)}} \left(M^{(k)}\right)^T \\ \frac{\partial J}{\partial Z^{(k)}} &= \frac{\partial J}{\partial a^{(k)}} \odot a^{(k)} \odot \left(1. - a^{(k)}\right) \\ \Longrightarrow \frac{\partial J}{\partial Z^{(k)}} &= \left(\frac{\partial J}{\partial Z^{(k+1)}} \left(W^{(k+1)}\right)^T \left(M^{(k)}\right)^T\right) \odot \left[a^{(k)} \odot \left(1. - a^{(k)}\right)\right] \end{aligned}$$

```
    dJ_dz2 = (a2 - y');
    dJ_dz1 = (dJ_dz2 * W2' * M1').* a1.* (1. - a1);
    % in function TwoLayerPerceptron.m &&
TwoLayerPerceptron_softmax.m
```

Thus

$$\frac{\partial J}{\partial W^{(k)}} = \left(A^{(k-1)}\right)^T \frac{\partial J}{\partial Z^{(k)}}$$

```
    dW1 = X * dJ_dz1; % X = A0'
    dW2 = A1'* dJ_dz2;
    % in function TwoLayerPerceptron.m &&
TwoLayerPerceptron_softmax.m
```

MATLAB implementation

TwoLayerPerceptron.m

```
function [W1, W2, a2, Cost] = TwoLayerPerceptron( X, y, W1_ini,
W2_ini, rho)
%[W1, W2, a2, Cost] = TwoLayerPerceptron( X, y, W1_ini, W2_ini)
% X: lxN    l = 3 (1, x1, x2)
% Y: 1xN
W1 = W1_ini; W2 = W2_ini;
[l, N] = size(X);
iter = 1; e = 1;
max_iter = 10000;
while (iter < max_iter) && (e > 0)
    z1 = X' * W1;
    a1 = sigmoid(z1); M1 = [zeros(size(a1, 2), 1), eye(size(a1,
2))];
    A1 = [ones(N, 1) a1];
    z2 = A1 * W2;
    a2 = sigmoid(z2); % a2: Probility of belonging to Class 1
    Loss = - y'.* log(a2) - (1. - y').* log(1. - a2);
    Cost = sum(Loss);
    % adjust weight
    dJ_dz2 = (a2 - y');
    dJ_dz1 = (dJ_dz2 * W2' * M1').* a1.* (1. - a1);
    dW1 = X * dJ_dz1;
    dW2 = A1'* dJ_dz2;
```

```
    W2 = W2 - rho * dW2;
    W1 = W1 - rho * dW1;
    e = sum( xor(y', (a2 > 0.5)) );
    iter = iter + 1;
end
fprintf('Number of iterations:\n')
fprintf([num2str(iter), '\n'])
```

TwoLayerPerceptron_softmax.m

```
function [W1, W2, a2, Cost] = TwoLayerPerceptron_softmax( X, y,
W1_ini, W2_ini, rho)
%[W1, W2, a2, Cost] = TwoLayerPerceptron_softmax( X, y, W1_ini,
W2_ini)
% X: lxN   l = 3 (1, x1, x2)
% y: size(a2, 2)xN
W1 = W1_ini; W2 = W2_ini;
[l, N] = size(X);
iter = 1; e = 1;
max_iter = 10000;
while (iter < max_iter) && ( e > 0 )
    z1 = X' * W1;
    a1 = sigmoid(z1); M1 = [zeros(size(a1, 2), 1), eye(size(a1,
2))];
    A1 = [ones(N, 1) a1];
    z2 = A1 * W2;
    a2 = softmax(z2);
    Cost = - trace( y * log(a2) );
    % adjust weight
    dJ_dz2 = (a2 - y');
    dJ_dz1 = (dJ_dz2 * W2' * M1').* a1.* (1. - a1);
    dW1 = X * dJ_dz1;
    dW2 = A1'* dJ_dz2;
    W2 = W2 - rho * dW2;
    W1 = W1 - rho * dW1;
    e = sum( sum( xor(y', (a2 > 0.5)) ) );
    iter = iter + 1;
end
fprintf('Number of iterations:\n')
fprintf([num2str(iter), '\n'])
```

Python implementation

backpropagate.py

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

def tanh(x):
    return np.tanh(x)

def tanh_prime(x):
    return 1 - x * x

def train(X, Y, lr=2., epoch=5000, err_break=1e-3,
          wo=np.ones((13, 1)), w1=np.ones((5, 1)), w2=np.ones((5,
1))):
    list_err = []
    for num_epoch in range(epoch):
        loss = 0
        for (x_tmp, y) in list(zip(X, Y)):
            len_input = len(w1) - 1
            len_hidden = len(x_tmp) - (len_input - 1)
            # Forward
            # v = [v1, v2, 1]; x = [..., 1]
            # net1: v1 = sigmoid(z1); z1 = x @ w1
            # net2: v2 = sigmoid(z2); z2 = x @ w2
            # out : o  = sigmoid(z ); z  = v @ wo
            # loss: sum( (y - o) * (y - o)/ 2. )
            x = []
            for start in range(len_hidden):
                end = start + len_input
                x.append(x_tmp[start:end])
            x  = np.hstack((np.asarray(x), np.ones((len(x), 1))))
            v1 = tanh(x @ w1)  # v1
            v2 = tanh(x @ w2)  # v2
            v = np.vstack((v1, v2))
            v = np.concatenate((v, [[1]]))
            o = tanh(v.transpose() @ wo)
            loss = loss + ((y - o) * (y - o))[0][0] / 2.
            # Backward
            # For o, v
            # d loss / d o = (o - y)
            # d loss / d v = (d z / d v)(d loss / d z)
```

```
            #               = wo * (d loss / d z)
            # For z, z1, z2
            # d o  / d z  = sigmoid_prime(o )
            # d v1 / d z1 = sigmoid_prime(v1)
            # d v2 / d z2 = sigmoid_prime(v2)
            #
            # d loss / d z
            # = (d o  / d z )(d loss / d o )
            # = (d o  / d z ) * (o - y)
            # = sigmoid_prime(o ) * (o - y)
            #
            # d loss / d z1
            # = (d v1 / d z1) * (d loss / d v)[1]
            # = sigmoid_prime(v1) * { wo * (d loss / d z) }[1]
            # = sigmoid_prime(v1) * wo[1] * (d loss / d z)
            #
            # d loss / d z2
            # = (d v2 / d z2) * (d loss / d v)[2]
            # = sigmoid_prime(v2) * { wo * (d loss / d z) }[2]
            # = sigmoid_prime(v2) * wo[2] * (d loss / d z)
            delta_zo = tanh_prime(o) * (o - y)  # d loss / d z
            delta_z1 = tanh_prime(v1) * (wo @ delta_zo)
[0:len_hidden]  # d loss / d z1
            delta_z2 = tanh_prime(v2) * (wo @ delta_zo)
[len_hidden:-1]  # d loss / d z2
            # For wo, w1, w2
            # d loss / d wo =  (d z  / d wo)(d loss / d z )
            #               = v  * (d loss / d z )
            # d loss / d w1 =  (d z1 / d w1)(d loss / d z1)
            #               = x' * (d loss / d z1)
            # d loss / d w2 =  (d z1 / d w2)(d loss / d z2)
            #               = x' * (d loss / d z2)
            delta_wo = v @ delta_zo  # d loss / d wo
            delta_w1 = x.transpose() @ delta_z1  # d loss / d w1
            delta_w2 = x.transpose() @ delta_z2  # d loss / d w2
            # update: w := w - lr * (d loss / d w)
            wo = wo - lr * delta_wo
            w1 = w1 - lr * delta_w1
            w2 = w2 - lr * delta_w2
        list_err.append(loss)
        if loss < err_break:
            print('Run ' + str(num_epoch) + ' epochs')
            break
```

```
    return wo, w1, w2, list_err


if __name__ == "__main__":
    X = np.array([
        [0, 0, 0.8, 0.4, 0.4, 0.1, 0, 0, 0],
        [0, 0.3, 0.3, 0.8, 0.3, 0, 0, 0, 0],
        [0, 0, 0, 0, 0.3, 0.3, 0.8, 0.3, 0],
        [0, 0, 0, 0, 0, 0.8, 0.4, 0.4, 0.1],
        [0.8, 0.4, 0.4, 0.1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0.3, 0.3, 0.8, 0.3],
    ])
    Y = np.asarray([[-1, 1, 1, -1, -1, 1]]).transpose()
    wo_init = np.asarray([[1.20973877, -1.07518386, 0.80691921,
-0.29078347, -0.22094764,
                           -0.16915604, 1.10083444, 0.08251052,
-0.00437558, -1.72255825,
                           1.05755642, -2.51791281,
-1.91064012]]).transpose()
    w1_init = np.asarray([[1.73673761, 1.89791391, -2.10677342,
-0.14891209, 0.58306155]]).transpose()
    w2_init = np.asarray([[-2.25923303, 0.13723954, -0.70121322,
-0.62078008, -0.47961976]]).transpose()
    wo, w1, w2, list_err = train(X, Y, lr=0.2, epoch=1000,
wo=wo_init, w1=w1_init, w2=w2_init)
    print('hidden layer 1, neuron 1 weights\n', w1)
    print('hidden layer 1, neuron 2 weights\n', w2)
    print('hidden layer 2, neuron 1 weights\n', wo)

    plt.plot(list_err)
    plt.ylabel('error')
    plt.xlabel('epochs')
    plt.show()
    for ind, (x_tmp, y) in enumerate(list(zip(X, Y))):
        len_input = len(w1) - 1
        len_hidden = len(x_tmp) - (len_input - 1)
        x = []
        for start in range(len_hidden):
            end = start + len_input
            x.append(x_tmp[start:end])
        x = np.hstack((np.asarray(x), np.ones((len(x), 1))))
        v1 = tanh(x @ w1)  # v1
        v2 = tanh(x @ w2)  # v2
```

```
        v = np.vstack((v1, v2))
        v = np.concatenate((v, [[1]]))
        o = tanh(sum(v * wo))
        print(str(ind) + ": produced: " + str(o) + " wanted " +
str(y))
```