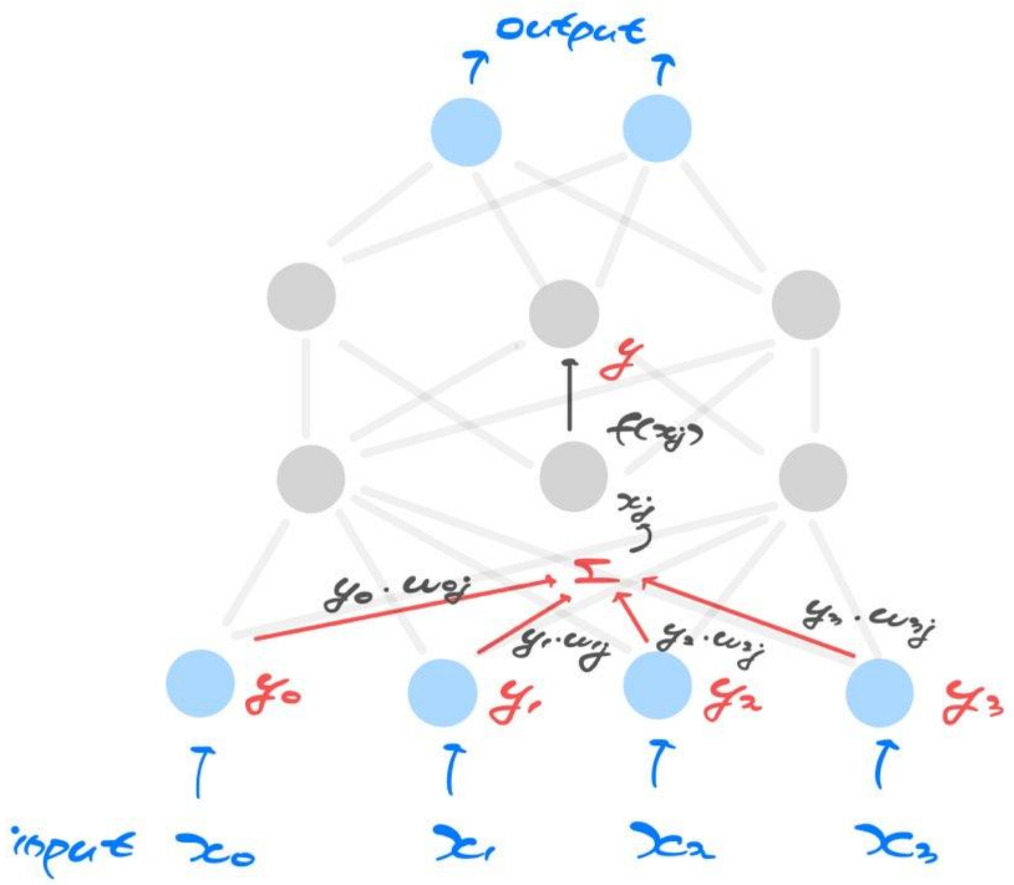
Learning representations by back-propagating errors

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams.

수학과 오서영

Neural Network



Back-propagation

If input units are directly connected to the output units

-> easy to find learning rules
(iteratively adjust the relative strengths)
-> progressively reduce the difference
between the actual and desired output

Hidden units

-> Learning procedure decides under what
circumstances the hidden units should be active in order to achieve the desired input-output
-> Hidden units are going to learn to represent some features of input domain
-> **Back propagation**

Feed Forward

Input $\boldsymbol{x_j}$

-> input of higher layer will be a weighted sum of the outputs of all of the lower layer units that feed into it
-> i feeds into j

Output $\boldsymbol{y_i}$
Adjusted by a weight on the link between i and j $\boldsymbol{w_{ij}}$

$$x_j = \sum y_i w_{ij}$$

Feed Forward

Output $\boldsymbol{y_j}$
-> **Activation function** in this paper is the **sigmoid function**

$$y_j = \frac{1}{1 + e^{-x_j}}$$

-> By repeating this procedure, starting with the **input layer**,
we can feed-forward through the network layers and arrive at a set of **outputs for the output layer**

Error Function

The aim is to find a set of weights that ensure that
the output vector is the same as the desired output vector

-> computed output value of j in output layer y_j
-> its desired state d_j

$$E = \frac{1}{2}\sum_{j}(y_j - d_j)^2$$

-> minimize
-> need to adjust the weights

Error Function

To minimize E by gradient descent it is necessary to compute the **partial derivative** of E with respect to each weight in the network

-> $\frac{\partial E}{\partial w}$ -> $\frac{\partial E}{\partial y_j}$ (E and y are directly related) -> $\frac{\partial y_j}{\partial x_j}$

$$\frac{\partial}{\partial y_j}\frac{1}{2}\sum_j (y_j - d_j)^2 = y_j - d_j \qquad y_j = \text{sigmoid}(x_j)$$

$$\frac{\partial y_j}{\partial x_j} = \text{sigmoid}(x_j) * (1 - \text{sigmoid}(x_j))$$

Error Function

$$x_j = \sum y_i w_{ij}$$

-> $\frac{\partial x_j}{\partial w_{ij}} = y_i$

We can chain them together to figure out how the error changes with the weights

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial y} * \frac{\partial y}{\partial x} * \frac{\partial x}{\partial w}$$

Gradient Descent

Simplest form of **gradient descent** would be to change each weight by an amount proportional to the accumulated gradient

$$\Delta \mathrm{w} = -\mathrm{a} * \frac{\partial E}{\partial w}$$

Code implementation

```
display(Math(r'x_j = ₩sum_{i=0}y_iw_{ij}'))
display(Math(r'y_j = ₩frac{1}{1+e^{-x_j}}'))
display(Math(r'E = ₩frac{1}{2}₩sum_{j}(y_j-d_j)^2'))
display(Math(r'₩frac{₩partial E}{₩partial y_j} = y_j - d_j'))
display(Math(r'₩frac{₩partial y_j}{₩partial x_j} = y_j*(1-y_j)'))
display(Math(r'₩frac{₩partial x_j}{₩partial w_{ij}} = y_i'))
```

$$x_j = \sum_{i=0} y_i w_{ij}$$

$$y_j = \frac{1}{1+e^{-x_j}}$$

$$E = \frac{1}{2}\sum_{j}(y_j-d_j)^2$$

$$\frac{\partial E}{\partial y_j} = y_j - d_j$$

$$\frac{\partial y_j}{\partial x_j} = y_j*(1-y_j)$$

$$\frac{\partial x_j}{\partial w_{ij}} = y_i$$

```
def sigmoid(xj):

    s = 1/(1+np.exp(-xj))

    return s
```

Code implementation

```
def propagate(w, yi, dj):

    m = yi.shape[1]

    yj = sigmoid(np.dot(w.T, yi))          # compute activation
    cost = 1/m*(np.sum(yj-dj))           # compute cost

    dw = 1/m*np.dot(yi, ((yj-dj)*yj*(1-yj)).T)

    cost = np.squeeze(cost)


    return yj, dw, cost
```

Code implementation

```
def optimize(w, yi, dj, num_iterations, learning_rate):

    costs = []

    for i in range(num_iterations):

        yj, dw, cost = propagate(w, yi, dj)

        w = w - learning_rate*dw

        if i % 1000 == 0:
            costs.append(cost)

        if i % 1000 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    return w, dw, yj, costs
```

Code implementation

```
w = np.zeros((2,3))
yi, dj = np.array([[1],[2]]), np.array([[1,0,1]]).T
print(yi.shape)
print(dj.shape)
```

```
(2, 1)
(3, 1)
```

```
w, dw, yj, costs = optimize(w, yi, dj, num_iterations= 10000, learning_rate = 0.1)

print ("w = " + str(w) + '₩n')
print ("dw = " + str(dw) + '₩n')
print("prediction = " + str(yj))
```

```
Cost after iteration 0: -0.007175
Cost after iteration 1000: -0.006999
Cost after iteration 2000: -0.006836
Cost after iteration 3000: -0.006684
Cost after iteration 4000: -0.006541
Cost after iteration 5000: -0.006407
Cost after iteration 6000: -0.006281
Cost after iteration 7000: -0.006162
Cost after iteration 8000: -0.006050
Cost after iteration 9000: -0.005943
w = [[ 1.02735774 -1.02735774  1.02735774]
 [ 2.05471549 -2.05471549  2.05471549]]

dw = [[-3.39330263e-05  3.39330263e-05 -3.39330263e-05]
 [-6.78660526e-05  6.78660526e-05 -6.78660526e-05]]

prediction = [[0.9941577]
 [0.0058423]
 [0.9941577]]
```