Back-propagation

강남웅

Back-Propagation

- 진행과정

  1. Forward propagation을 통해 initial value로 각 node의 값을 계산한다

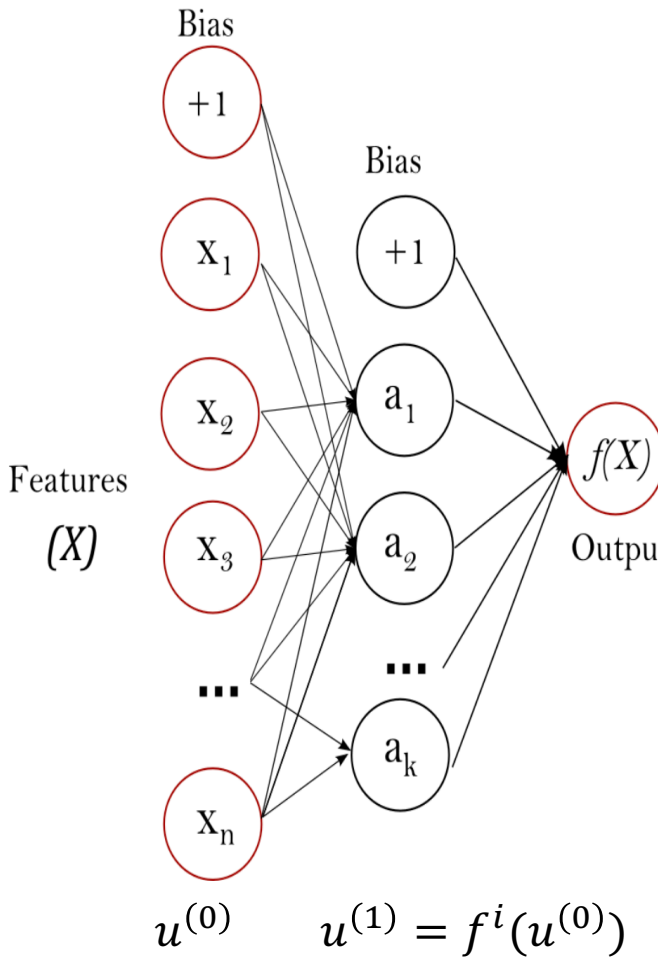
  2. Backward propagation을 사용해 derivative of output w.r.t input variable를 구한다

  3. 각 Layer의 weight를 gradient * step_size만큼 update를 해준다

Algorithm 6.1

Algorithm 6.1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}$, $j < i$, with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$



Algorithm 6.2

Algorithm 6.2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ with respect to the variables in the graph. This example is intended to further understanding by showing a simplified case where all variables are scalars, and we wish to compute the derivatives with respect to $u^{(1)}, \ldots, u^{(n_i)}$. This simplified version computes the derivatives of all nodes in the graph. The computational cost of this algorithm is proportional to the number of edges in the graph, assuming that the partial derivative associated with each edge requires a constant time. This is of the same order as the number of computations for the forward propagation. Each $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is a function of the parents $u^{(j)}$ of $u^{(i)}$, thus linking the nodes of the forward graph to those added for the back-propagation graph.

Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network

Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table`$[u^{(i)}]$ will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

`grad_table`$[u^{(n)}] \leftarrow 1$

for $j = n - 1$ down to 1 **do**

  The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:

  `grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return {`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i$}

예시

$$\frac{\partial \mathbf{Y}}{\partial x} = \begin{bmatrix} \frac{\partial y_{11}}{\partial x} & \frac{\partial y_{12}}{\partial x} & \cdots & \frac{\partial y_{1n}}{\partial x} \\ \frac{\partial y_{21}}{\partial x} & \frac{\partial y_{22}}{\partial x} & \cdots & \frac{\partial y_{2n}}{\partial x} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_{m1}}{\partial x} & \frac{\partial y_{m2}}{\partial x} & \cdots & \frac{\partial y_{mn}}{\partial x} \end{bmatrix}.$$

295 × 154

각 column이 $\frac{du^{(n)}}{du^{(i)}}$ 을 뜻한다고 생각하면

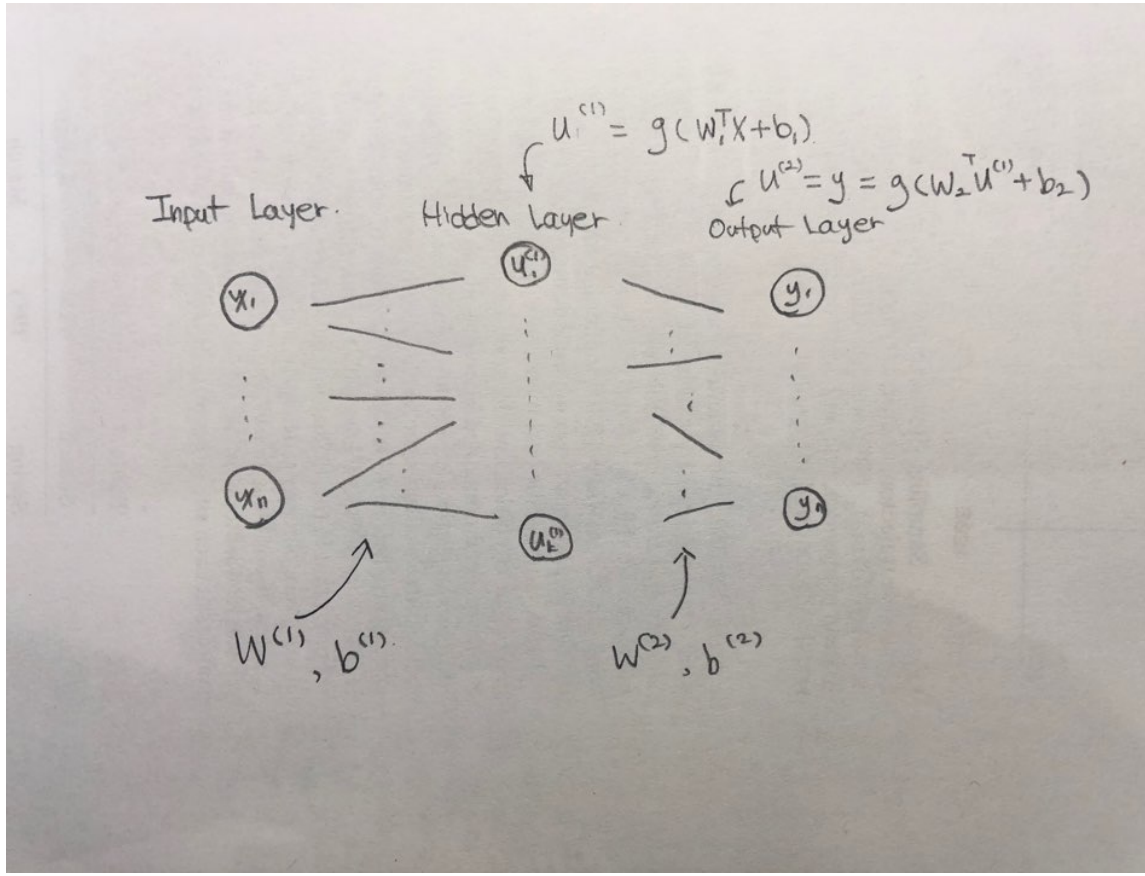
왼쪽의 grad_table과 같은 구조가 된다

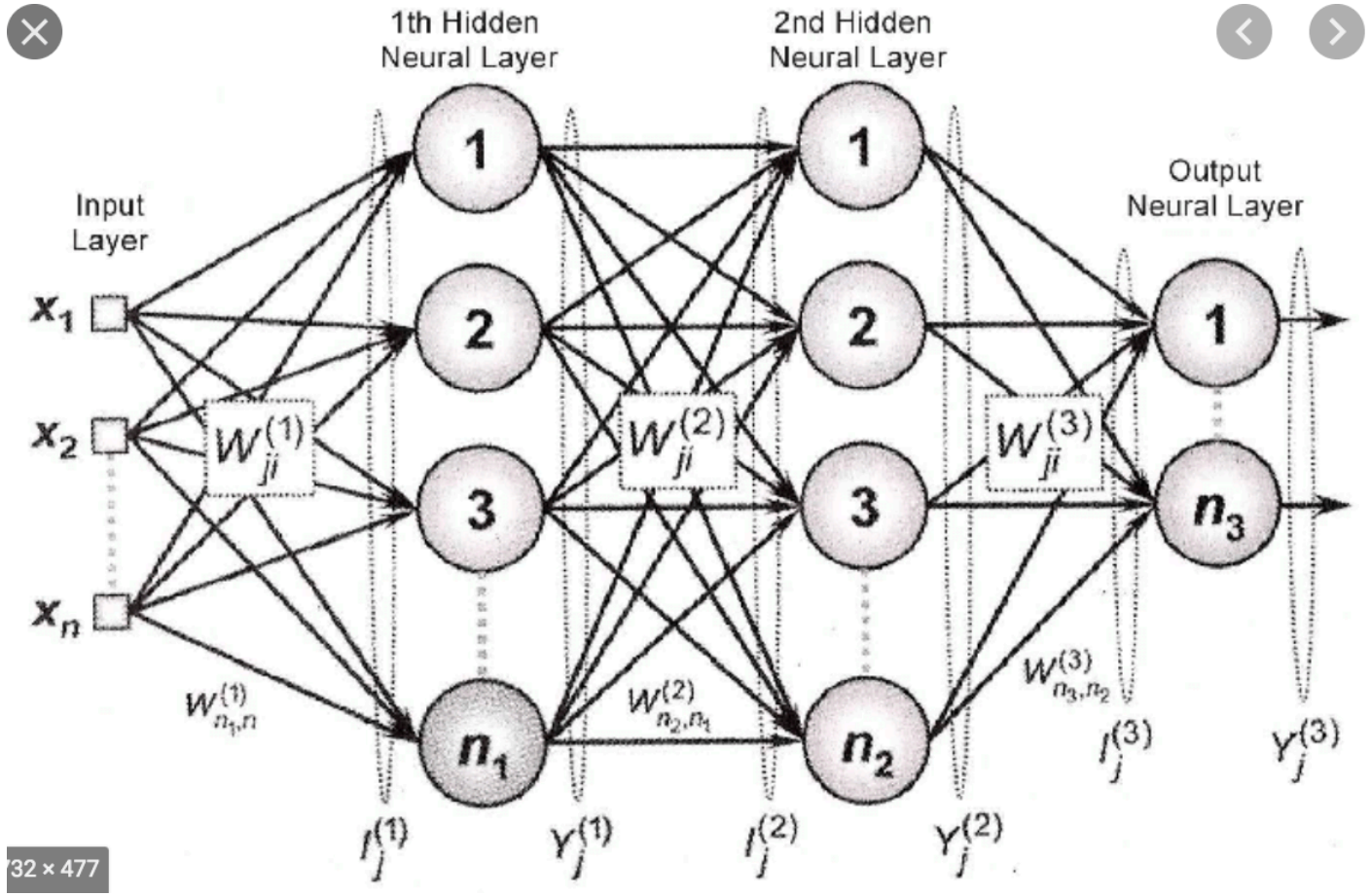
Algorithm 6.3

Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$




1th Hidden
Neural Layer
2nd Hidden
Neural Layer
Input
Layer
Output
Neural Layer
x_1
x_2
x_n
1
2
3
n_1
1
2
3
n_2
1
n_3
$W_{ji}^{(1)}$
$W_{ji}^{(2)}$
$W_{ji}^{(3)}$
$W_{n_1,n}^{(1)}$
$W_{n_2,n_1}^{(2)}$
$W_{n_3,n_2}^{(3)}$
$I_j^{(1)}$
$Y_j^{(1)}$
$I_j^{(2)}$
$Y_j^{(2)}$
$I_j^{(3)}$
$Y_j^{(3)}$

732 × 477

Algorithm 6.4

Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses in addition to the input $\boldsymbol{x}$ a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

Compute gradients on weights and biases (including the regularization term, where needed):

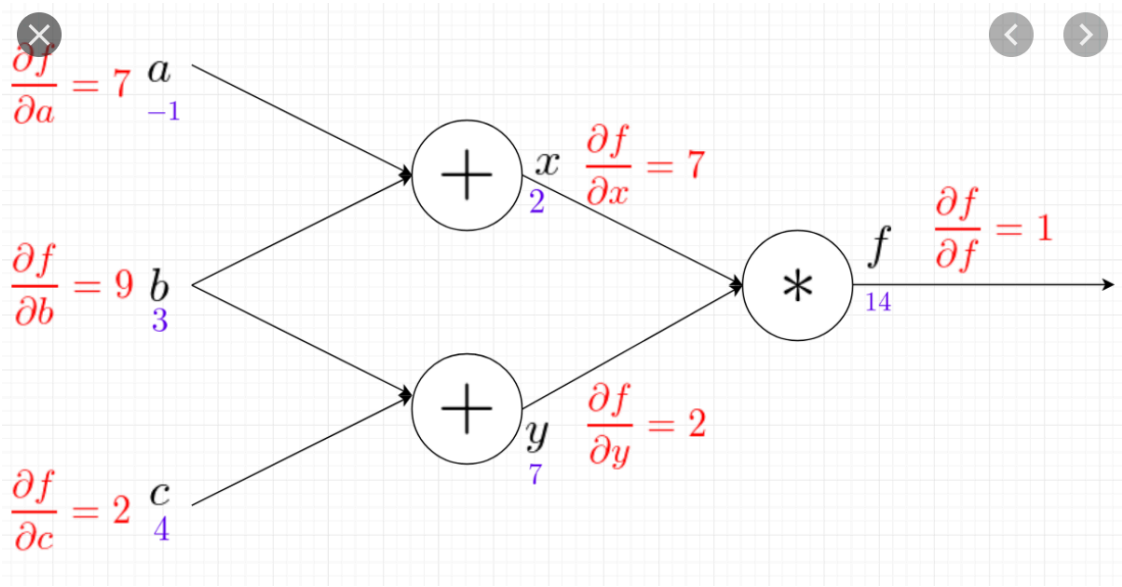
$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

end for



Code Example

- Add Layer
  - 주어진 input들의 sum을 구하는 Layer
  - forward prop
    - 주어진 input들을 저장
    - sum을 return
  - backward prop
    - local derivative가 1이므로 그대로 return

```python
# Perform Add(x, y) = x + y
class AddLayer(BinaryInputLayer):
    def forward(self, x, y):
        self.x = x
        self.y = y
        output = x + y

        return output

    @staticmethod
    def backward(self, dout):
        dx = dout
        dy = dout

        return [dx, dy]
```

Code Example

- Multiply Layer
  - 주어진 input들의 곱을 구하는 Layer
  - forward prop
    - 주어진 input 저장
      (backprop에서 사용)
    - 곱을 return
  - backward prop
    - local derivative가 각각 x, y이므로

      곱해주고 return

```python
# Perform Mul(x, y) = x * y
class MultiplyLayer(BinaryInputLayer):
    def forward(self, x, y):
        self.x = x
        self.y = y
        output = x * y

        return output

    def backward(self, dout):
        dx = dout * self.y
        dy = dout * self.x

        return [dx, dy]
```

Code Example

- ReLu Layer
  - max(0, input)을 수행하는 Layer
  - forward prop
    - 변수 저장
    - mask를 사용해서 return
  - backward prop
    - 주어진 derivative에서 0으로 rectified 된 원소는 0으로 변환 후 return

```python
# Perform ReLu(x) = max(0, x)
class ReLuLayer(BinaryInputLayer):
    def forward(self, x):
        self.x = x
        # use self.y as cache
        # vector/matrix of Boolean (mask)
        # ex. [[True False] [False False]]
        self.y = (x <= 0)
        # Replace neg val to 0
        output = x.copy()
        output[self.y] = 0

        return output

    def backward(self, dout):
        dx = dout
        # Replace neg val to 0
        # neg val has no impact on output
        dx[self.y] = 0

        return dx
```

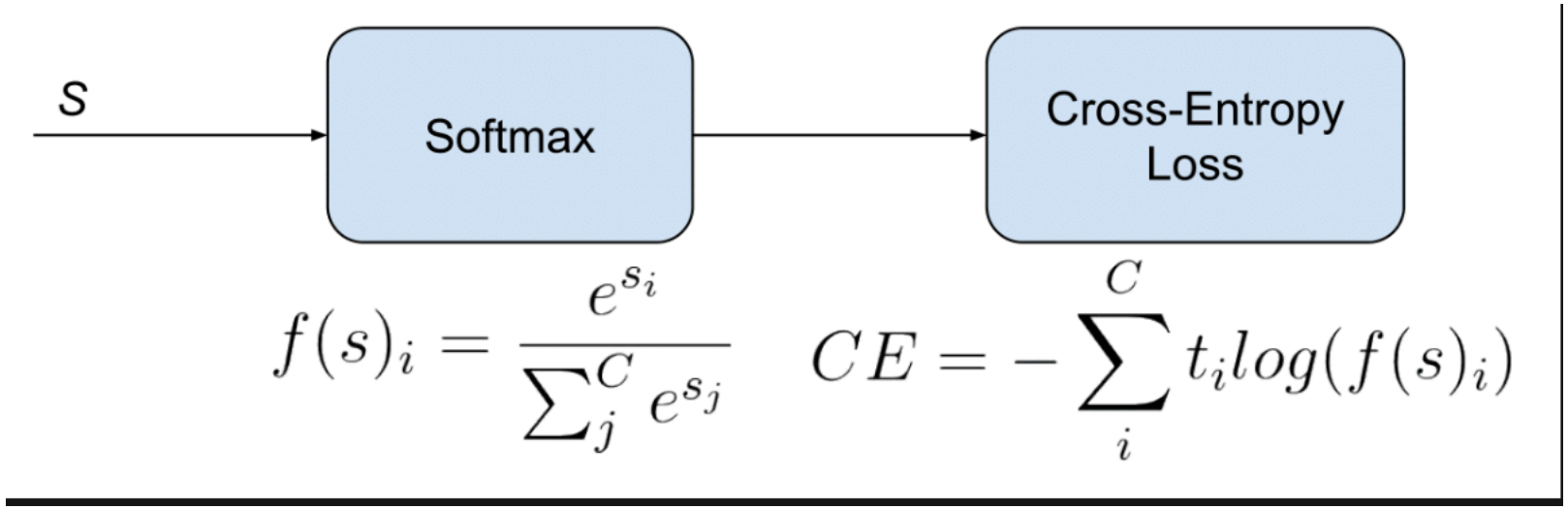
Code Example

- Sigmoid Layer
  - forward prop
    - 변수 저장
    - Sigmoid(x)를 return
      (여기서, exp(-x)를 사용하게 되면, overflow가 발생할 수 있으므로 x – max(x)로 교체하여 계산하는 것이 좋다)
  - backward prop
    - sigmoid_prime(x) = sigmoid(x) * (1 – sigmoid(x))

```python
# Perform Sigmoid(x) = 1 / 1 + exp(-x)
class SigmoidLayer(BinaryInputLayer):
    def forward(self, x):
        self.x = x
        # TODO : np.exp() is not stable because of Inf val. ( ex. exp(1000
        # np.exp(-x) -> np.exp(x - x.max())
        output = 1 / (1 + np.exp(-x))
        # use self.y as cache (for backprop)
        # Derivative of d Sigmoid(x) / dx = Sigmoid(x) * ( 1 - Sigmoid(x)
        # self.y = Sigmoid(x)
        self.y = output

        return output

    def backward(self, dout):
        dx = dout * self.y * (1 - self.y)

        return dx
```

Code Example

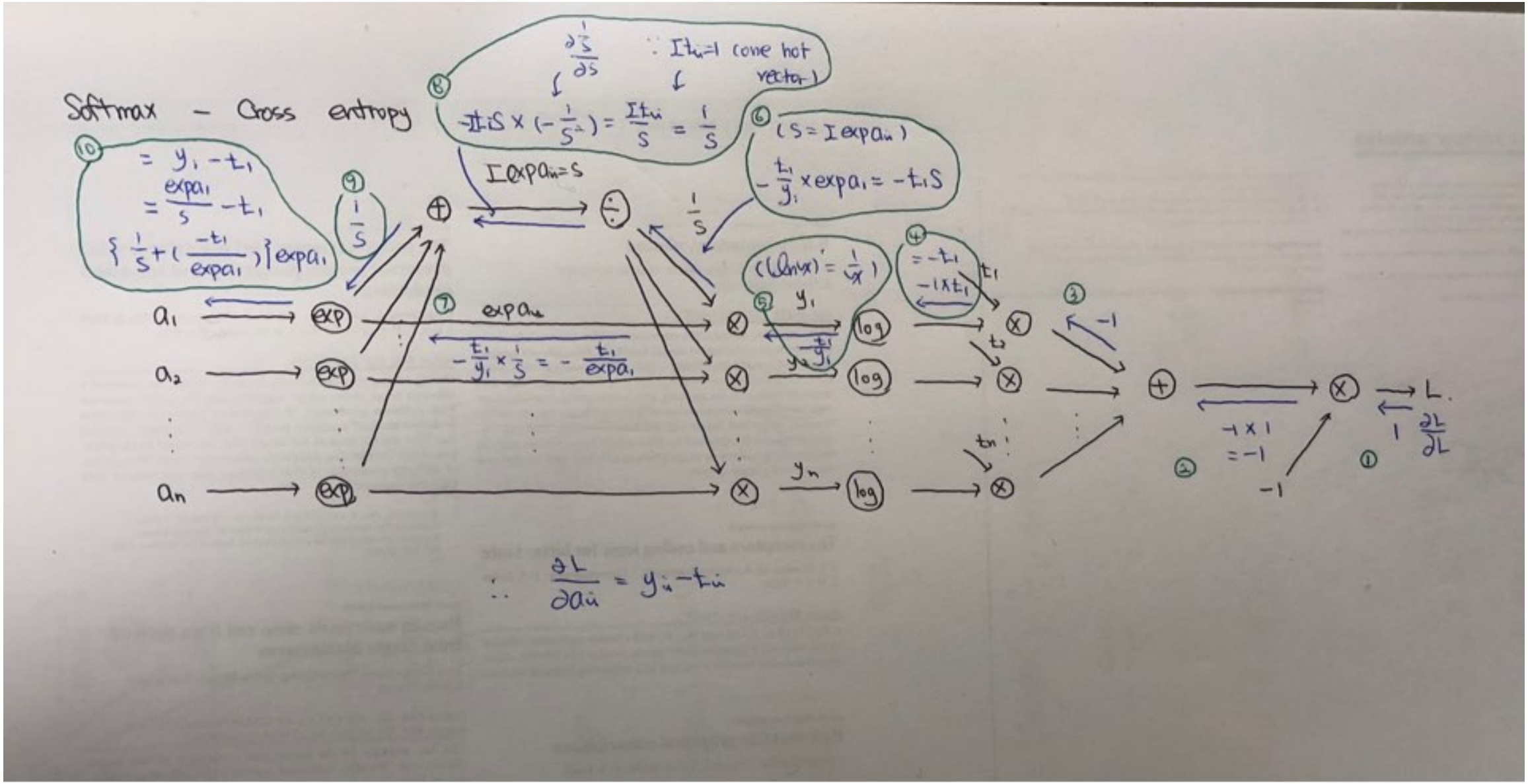
- Softmax with Loss



- Softmax를 구하고 난 뒤, cross-entropy loss를 구하는 Layer
  (이 때, t는 one hot vector)

Code Example

- Backward prop


Softmax - Cross entropy
$\frac{\partial \frac{1}{S}}{\partial S}$
$\because I t_i = 1$ (one hot vector)
$-\frac{1}{S} \times (-\frac{1}{S^2}) = \frac{I t_i}{S} = \frac{1}{S}$
$S = I \exp a_i$
$-\frac{t_i}{y_i} \times \exp a_i = -t_i S$
$I \exp a_i = S$
$= y_i - t_i$
$= \frac{\exp a_i}{S} - t_i$
$\{\frac{1}{S} + (\frac{-t_i}{\exp a_i})\} \exp a_i$
$\frac{1}{S}$
$(\log x)' = \frac{1}{x}$
y_i
$= -t_i$
$-1 \times t_i$
t_i
$\exp a_i$
$-\frac{t_i}{y_i} \times \frac{1}{S} = -\frac{t_i}{\exp a_i}$
$\frac{t_i}{y_i}$
a_1
a_2
a_n
exp
log
y_n
t_n
$-1 \times 1 = -1$
L
$\frac{\partial L}{\partial L}$
$\therefore \frac{\partial L}{\partial a_i} = y_i - t_i$

Code Example

- backward prop
  - $dx = y_i - t_i \quad where\ y_i = self.x$
    $t_i = self.y$
  - 각 변수에 대한 derivative를 return 하기 위해 batch_size로 나누고 return

```python
class SoftMaxLossLayer(BinaryInputLayer):

    @staticmethod
    def softmax(z):
        exp_z = np.exp(z)
        y = exp_z / np.sum(exp_z)

        return y

    @staticmethod
    def cross_entropy(y, t):
        return -np.sum(t * np.log(y))

    def forward(self, x, y):
        # input x
        self.x = self.softmax(x)
        # one-hot encoding t
        self.y = y

        loss = self.cross_entropy(self.x, self.y)

        return loss

    def backward(self, dout):
        batch_size = self.y.shape[0]
        dx = (self.x - self.y) / batch_size

        return dx
```

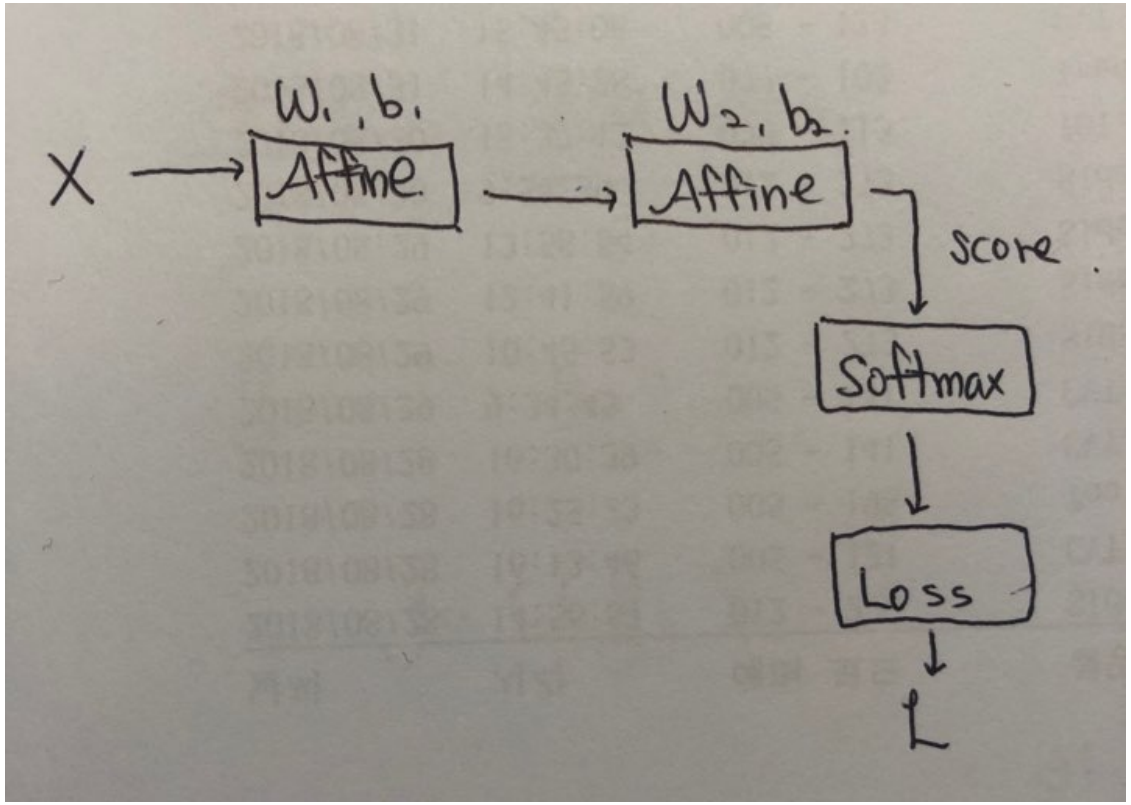
Code Example

- Multi Class Classification using Gradient Decent
  - Model
    - $Input\ X$
    - Layer#1

      $Affine\ Layer\ (\ with\ W_1, b_1\ )$
    - Layer#2

      $Affine\ Layer\ (\ with\ W_2, b_2\ )$
    - Layer#3

      $Softmax\ Layer$



Code Example

- Multi Class Classification using Gradient Decent
  - 데이터 생성

```
##############################################################################
# DATA GENERATION
##############################################################################

# number of data points per class
N = 100
# dimensionality
D = 2
# number of classes
K = 3

# Data
X = np.zeros((N*K,D))
# Label
y = np.zeros(N*K, dtype='uint8')

for j in range(K):
  ix = range(N*j,N*(j+1))
  r = np.linspace(0.0,1,N) # radius
  t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
  X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
  y[ix] = j

# Data visualization
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
plt.show()

##############################################################################
```

Code Example

- Multi Class Classification using Gradient Decent
  - 파라미터 생성

```
# Parameter GENERATION

h = 100
# np.random.randn => Generate Random number from N(0,1)
W = 0.01 * np.random.randn(D, h)
b = np.zeros((1, h))
W2 = 0.01 * np.random.randn(h, K)
b2 = np.zeros((1, K))

step_size = 1e-0
```

Code Example

- Multi Class Classification using Gradient Decent
  - Gradient Decent Loop (Pt.1)

```
# Create Two Affine Layer
hidden_l = AffineLayer(W, b)
score_l = AffineLayer(W2, b2)

# Execute Foward Propagation
out_1 = hidden_l.forward(X)
out_2 = score_l.forward(out_1)

# Provide score(out_2) to SoftMax func
exp_l = np.exp(out_2)
probs_l = exp_l / np.sum(exp_l, axis=1, keepdims=True)

# Extract Probability of real class
correct_logprobs_l = -np.log(probs_l[range(num_examples), y])
data_loss_l = np.sum(correct_logprobs_l) / num_examples
loss_l = data_loss_l
if i % 1000 == 0:
    print("Custom iteration %d: loss %f" % (i, loss_l))
```

Code Example

- Multi Class Classification using Gradient Decent
  - Gradient Decent Loop (Pt.2)

```
# Calculate Gradients
dscores_l = probs_l
dscores_l[range(num_examples), y] -= 1
dscores_l /= num_examples

dhidden_l = score_l.backward(dscores_l)
dW2_l = score_l.dW
db2_l = score_l.db
dhidden_l[out_1 <= 0] = 0
hidden_l.backward(dhidden_l)
dW_l = hidden_l.dW
db_l = hidden_l.db
```

Code Example

- Multi Class Classification using Gradient Decent
  - Gradient Decent Loop (Pt.3)

```
# Update Parameters
W += -step_size * dW_l
b += -step_size * db_l
W2 += -step_size * dW2_l
b2 += -step_size * db2_l
```

Code Example

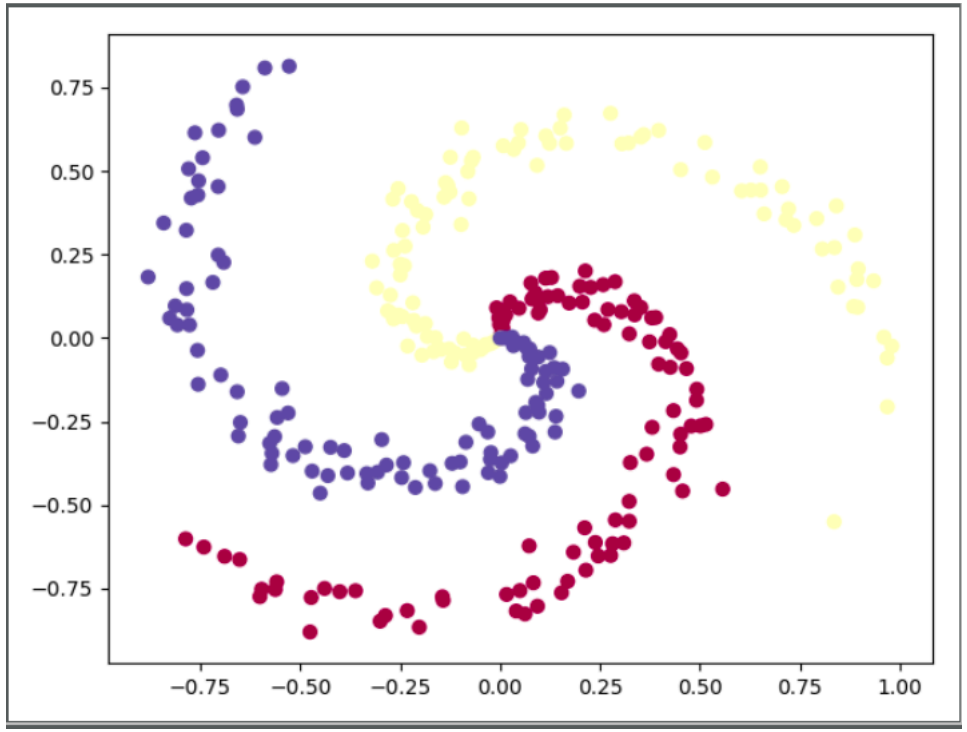
- Multi Class Classification using Gradient Decent

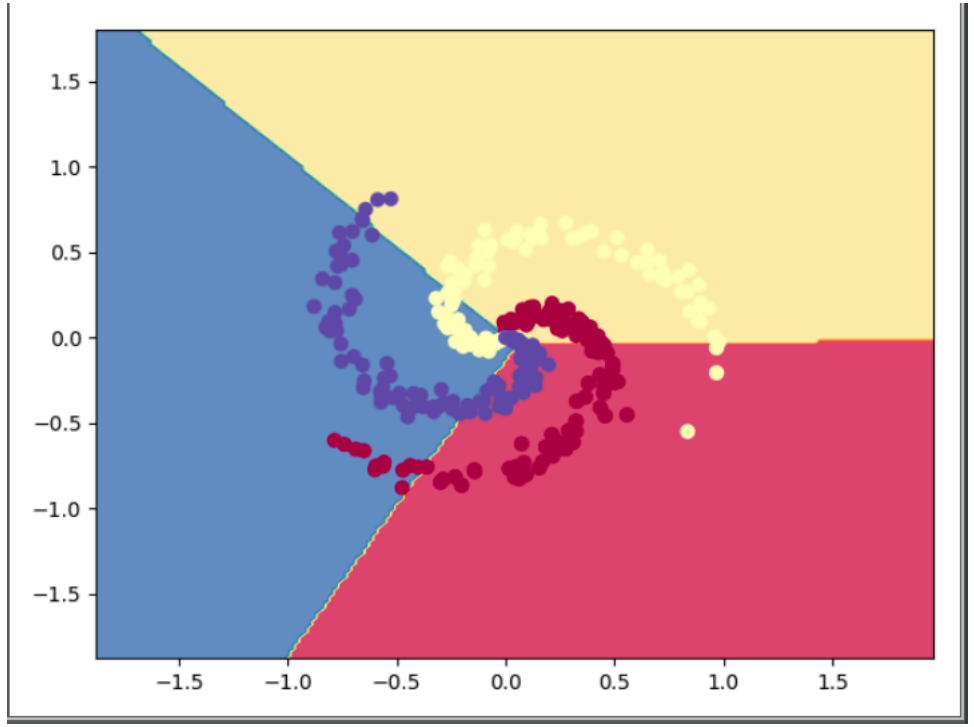
  - Gradient Decent Loop (Entire Code)

```
num_examples = X.shape[0]
for i in range(10000):
    # Create Two Affine Layer
    hidden_l = AffineLayer(W, b)
    score_l = AffineLayer(W2, b2)

    # Execute Foward Propagation
    out_1 = hidden_l.forward(X)
    out_2 = score_l.forward(out_1)

    # Provide score(out_2) to SoftMax func
    exp_l = np.exp(out_2)
    probs_l = exp_l / np.sum(exp_l, axis=1, keepdims=True)

    # Extract Probability of real class
    correct_logprobs_l = -np.log(probs_l[range(num_examples), y])
    data_loss_l = np.sum(correct_logprobs_l) / num_examples
    loss_l = data_loss_l
    if i % 1000 == 0:
        print("Custom iteration %d: loss %f" % (i, loss_l))

    # Calculate Gradients
    dscores_l = probs_l
    dscores_l[range(num_examples), y] -= 1
    dscores_l /= num_examples

    dhidden_l = score_l.backward(dscores_l)
    dW2_l = score_l.dW
    db2_l = score_l.db
    dhidden_l[out_1 <= 0] = 0
    hidden_l.backward(dhidden_l)
    dW_l = hidden_l.dW
    db_l = hidden_l.db

    # Update Parameters
    W += -step_size * dW_l
    b += -step_size * db_l
    W2 += -step_size * dW2_l
    b2 += -step_size * db2_l
```

Code Example

- Multi Class Classification using Gradient Decent
  - 결과





Implementation of library

- TensorFlow
  - GradientDescentOptimizer

```
# Minimize
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
train = optimizer.minimize(cost)
```

    - 직접 구현하는 것과 다르게 이미 선언된 Variable들에 대해 minimize (https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/GradientDescentOptimizer)

Implementation of library

- class GradientDescentOptimizer(optimizer.Optimizer)
  - https://github.com/tensorflow/tensorflow/blob/e5bf8de410005de06a7ff5393fafdf832ef1d4ad/tensorflow/python/training/gradient_descent.py#L30
- class Optimizer -> func minimize()
  - https://github.com/tensorflow/tensorflow/blob/e5bf8de410005de06a7ff5393fafdf832ef1d4ad/tensorflow/python/training/optimizer.py#L355

Implementation of library

- Layer Implementation of Library
  - Tensorflow
    - ReLu Layer
      - https://github.com/tensorflow/tensorflow/blob/v2.1.0/tensorflow/python/keras/layers/advanced_activations.py#L273-L332
  - Caffe
    - Sigmoid Layer
      - https://github.com/BVLC/caffe/blob/master/src/caffe/layers/sigmoid_layer.cpp
  - Torch
    - Mul Layer
      - https://github.com/torch/nn/blob/master/MulConstant.lua