Deep Learning Study Week #4

강준하

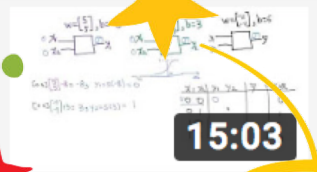
Fast Review

- Learning rate, Overfitting, Regularization
- Training set / Cross-validation set / Test set

- Perceptron 등장
- XOR Problem (1st winter of deep learning)
  → Backpropagation
- Gradient Vanishing Problem (2nd winter of deep learning)
  → ReLU (다음 주…)

모두를 위한 딥러닝 강좌 시즌 1

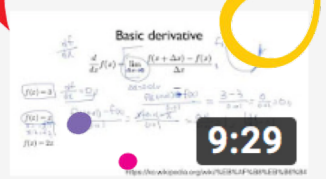
Sung Kim - 25 / 50



lec9-1: XOR 문제 딥러닝으로 풀기

Sung Kim

26



lec9-x: 특별편: 10분안에 미분 정리하기 (lec9-2 이전에 보세요)

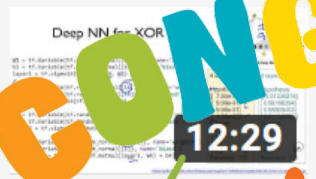
Sung Kim

27

lec9-2: 딥넷트웍 학습 시키기 (backpropagation)

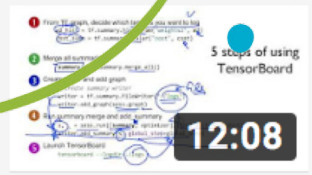
Sung Kim

28



Sung Kim

29



ML lab 09-2: Tensorboard (Neural Net for XOR)

Sung Kim

30

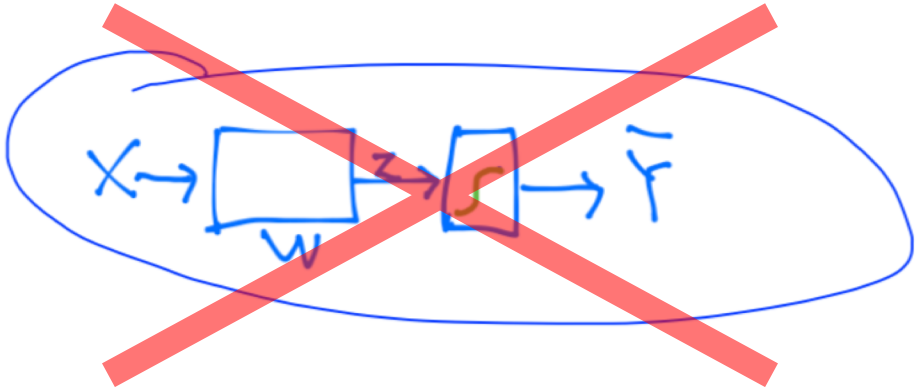
lec10-1: Sigmoid 보다 ReLU가 더 좋아

Sung Kim

CONGRATULATIONS

Lec 9-1

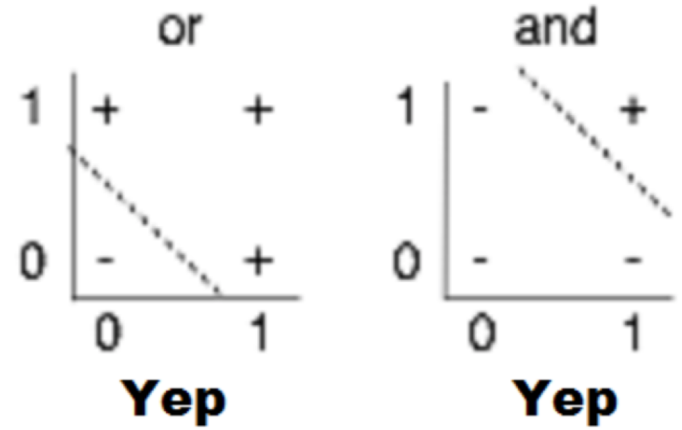
- 한 개의 Logistic Regression unit으로는 XOR 문제를 풀지 못했다…



Lec 9-1

Motivation

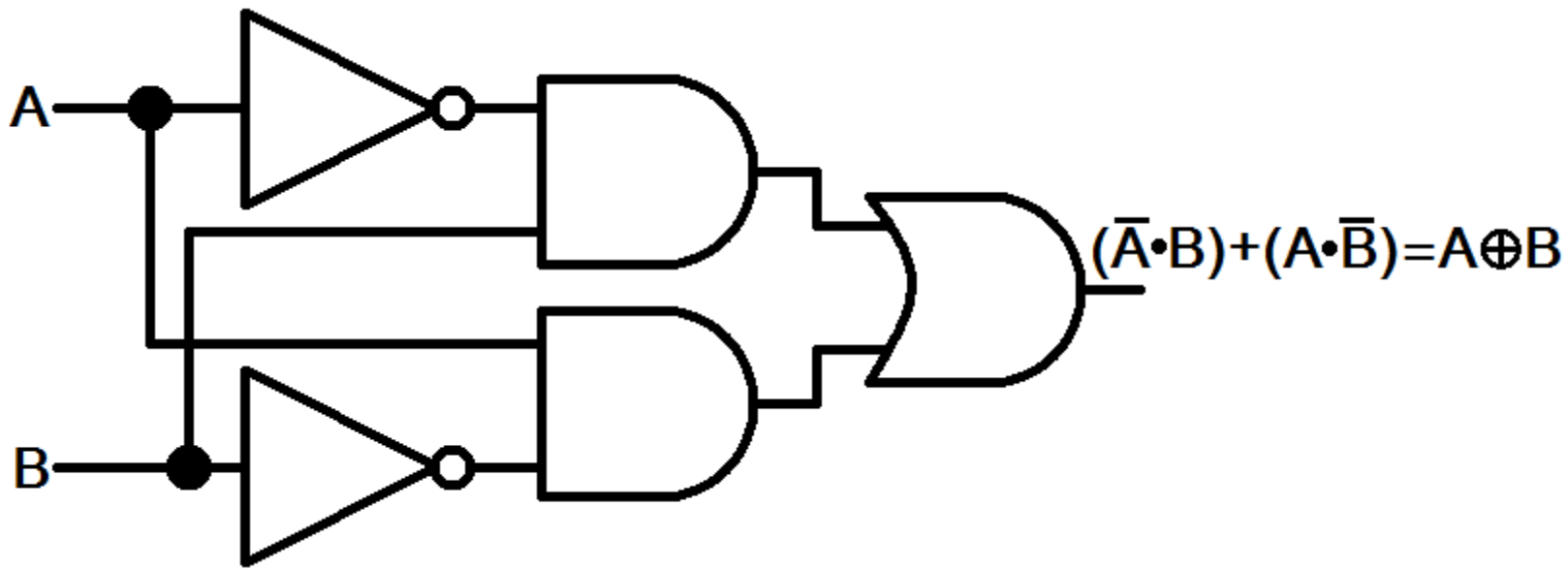
- AND gate와 OR gate는 하나의 regression unit으로도 만들 수 있다



- AND gate와 OR gate를 여러개 이용하면 XOR을 만들 수 있다...?

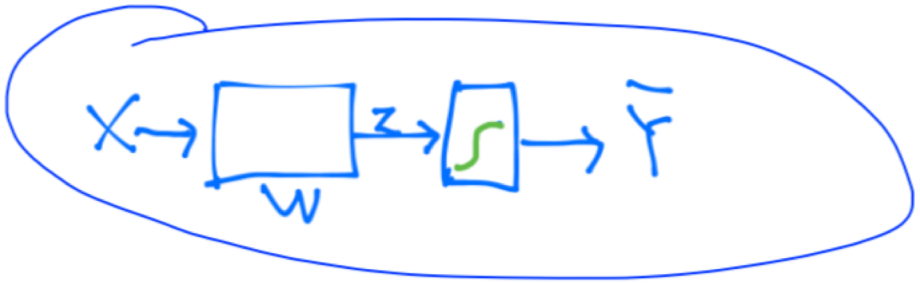
Lec 9-1

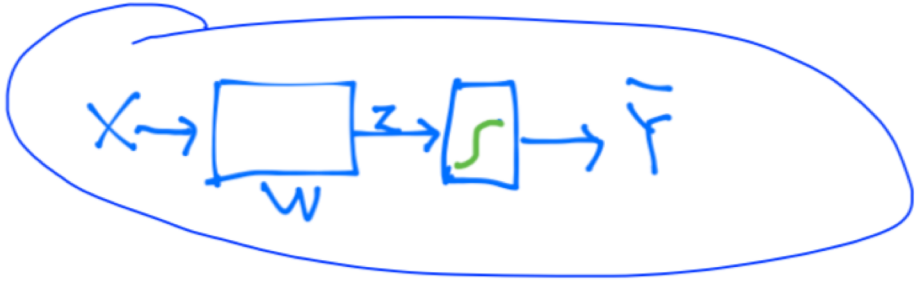
- XOR gate by AND gate and OR gate

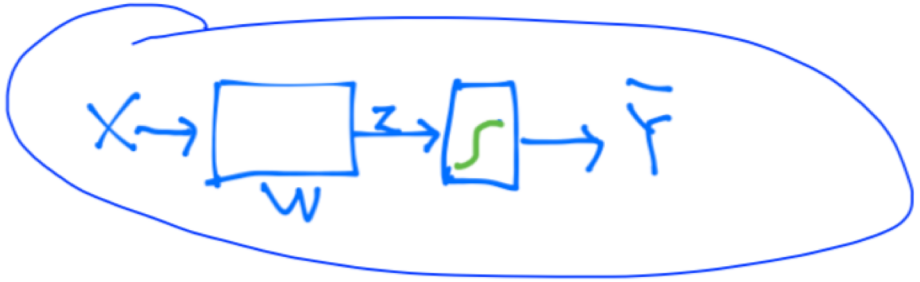


Lec 9-1

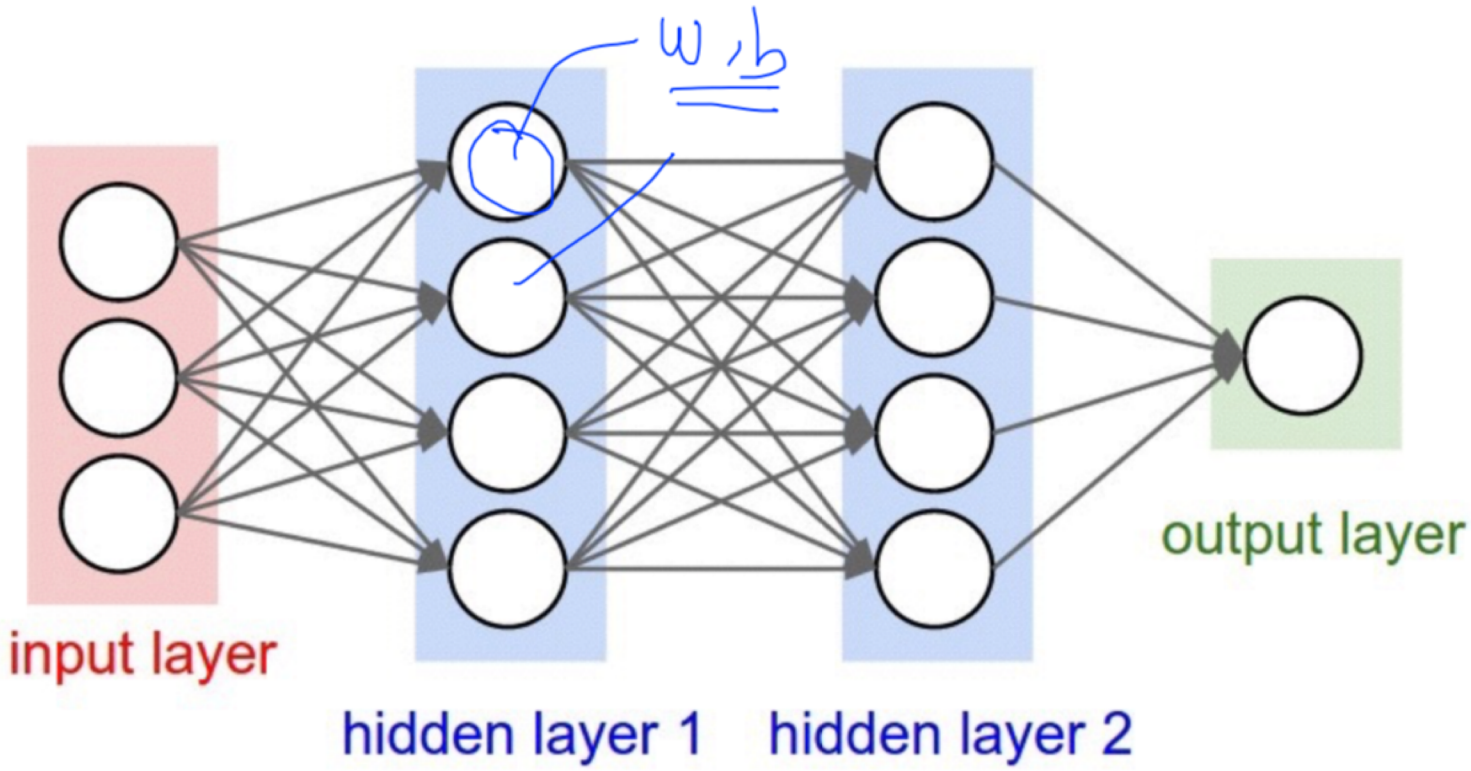
- Use multiple regression units!



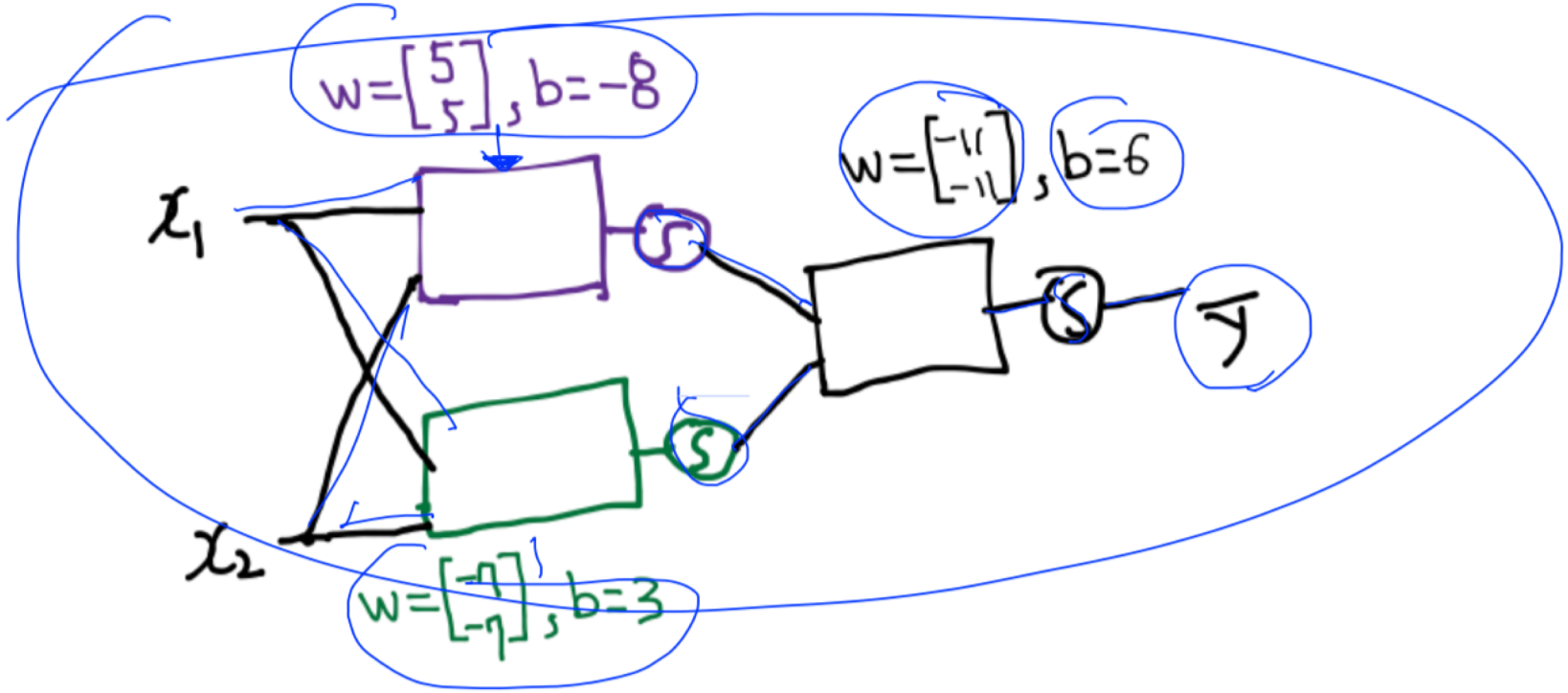




Lec 9-1



Lec 9-1



Lec 9-1

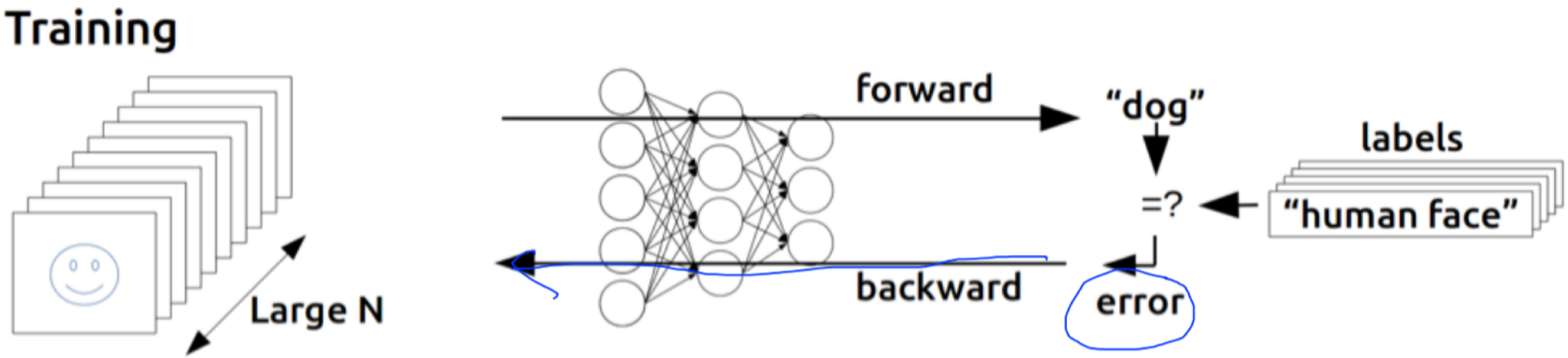
- How to learn multiple W and b?

Lec 9-X (미분 특별편)

- 스킵

Lec 9-2

- How to learn multiple W and b?
- Backpropagation!



Lec 9-2

- 미분? 각각의 노드가 결과값에 얼마나/어떻게 영향을 미치는지를 알기 위한 수단!

- Chain Rule? 뒤에서부터 앞으로 거슬러가며 미분값을 계산하는 방법!

Back propagation (chain rule)

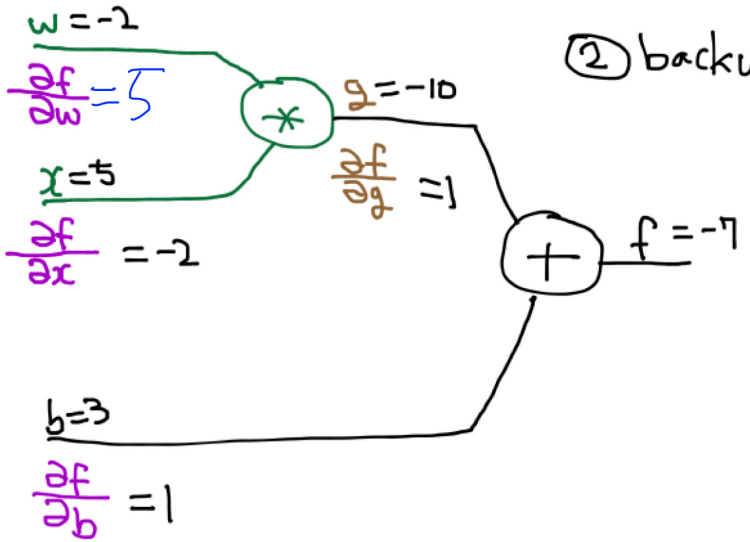
$f = wx + b$, $g = wx$, $f = g + b$

$\frac{\partial f}{\partial g} = 1$, $\frac{\partial f}{\partial b} = 1$

$\frac{\partial g}{\partial w} = x$, $\frac{\partial g}{\partial x} = w$

① forward ($w = -2, x = 5, b = 3$)

② backward

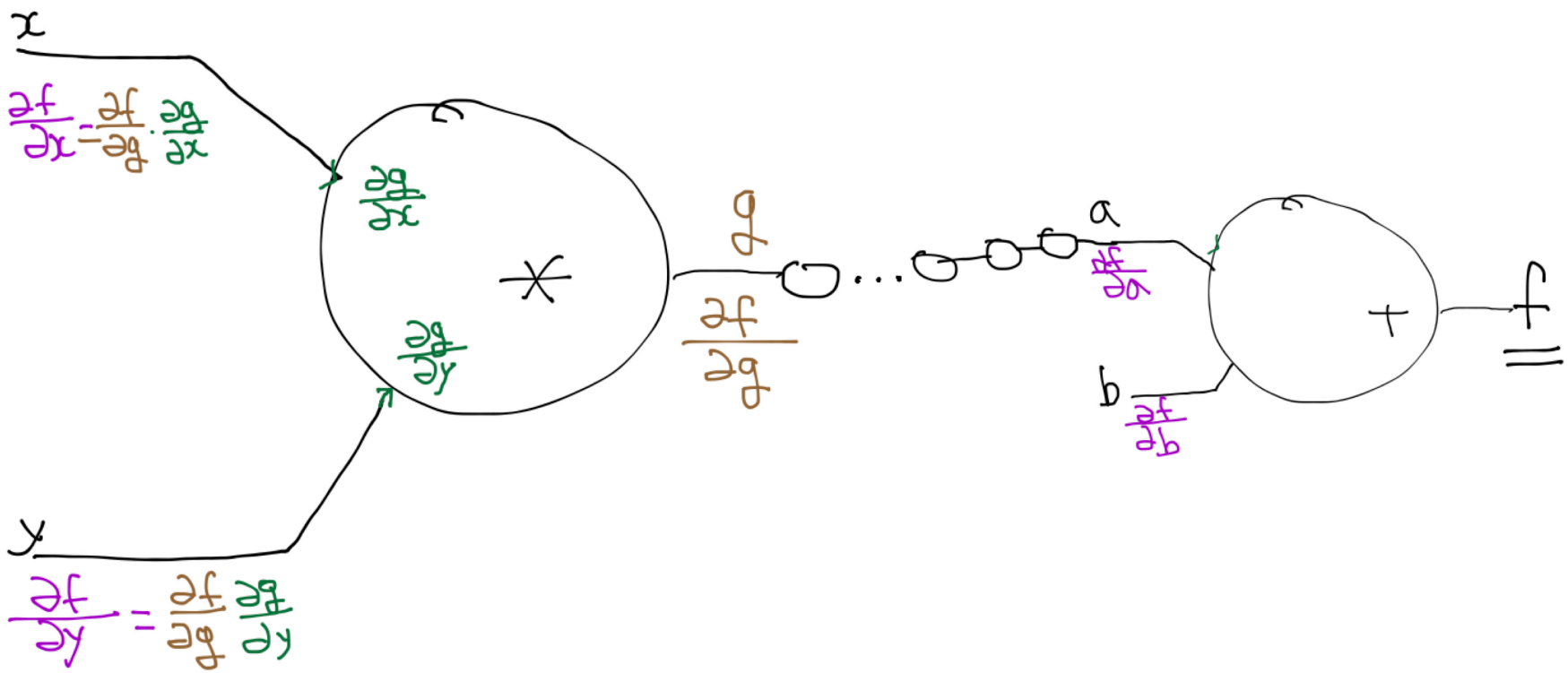


$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} = 1 * w = -2$

$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} = 1 * x = 5$

http://cs231n.stanford.edu/

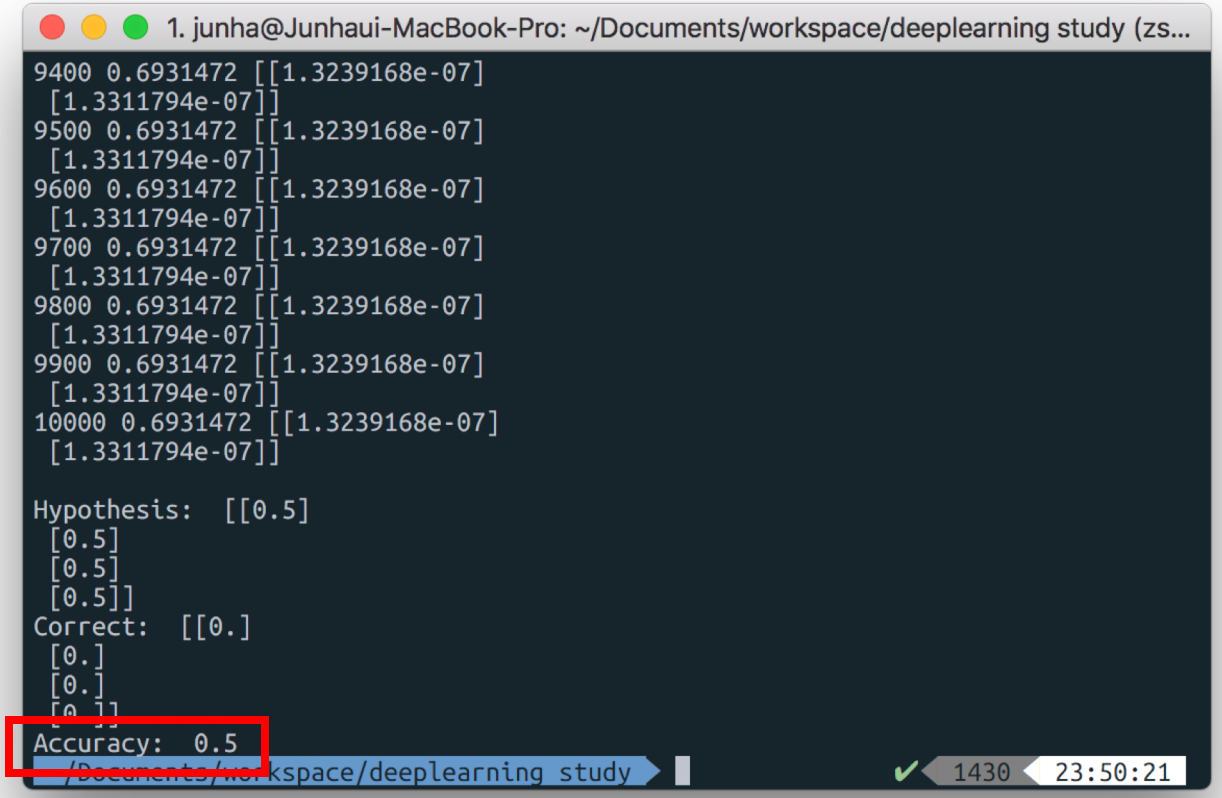
Back propagation (chain rule)



http://cs231n.stanford.edu/

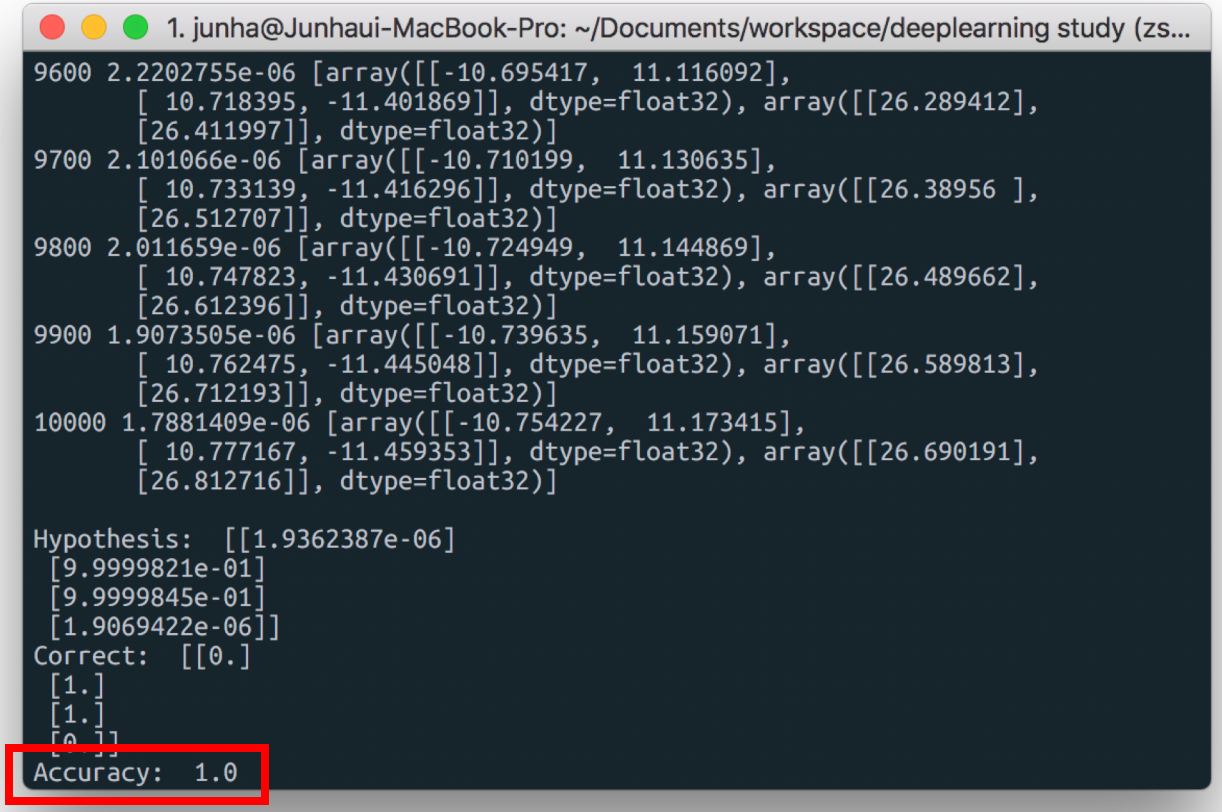
Lab 9-1

- 한 개의 Logistic Regression Unit을 사용했을 경우



Lab 9-1

- Multiple Logistic Regression Unit을 사용했을 경우



Lab 9-1

- Multilayer structure

```
W1 = tf.Variable(tf.random_normal([2, 2]), name='weight1')
b1 = tf.Variable(tf.random_normal([2]), name='bias1')
layer1 = tf.sigmoid(tf.matmul(X, W1) + b1)

W2 = tf.Variable(tf.random_normal([2, 1]), name='weight2')
b2 = tf.Variable(tf.random_normal([1]), name='bias2')
hypothesis = tf.sigmoid(tf.matmul(layer1, W2) + b2)
```

Lab 9-1

- Multilayer structure
- More **wider**…

```
W1 = tf.Variable(tf.random_normal([2, 10]), name='weight1')
b1 = tf.Variable(tf.random_normal([10]), name='bias1')
layer1 = tf.sigmoid(tf.matmul(X, W1) + b1)

W2 = tf.Variable(tf.random_normal([10, 1]), name='weight2')
b2 = tf.Variable(tf.random_normal([1]), name='bias2')
hypothesis = tf.sigmoid(tf.matmul(layer1, W2) + b2)
```

Lab 9-1

- Multilayer structure
- More **deeper**…

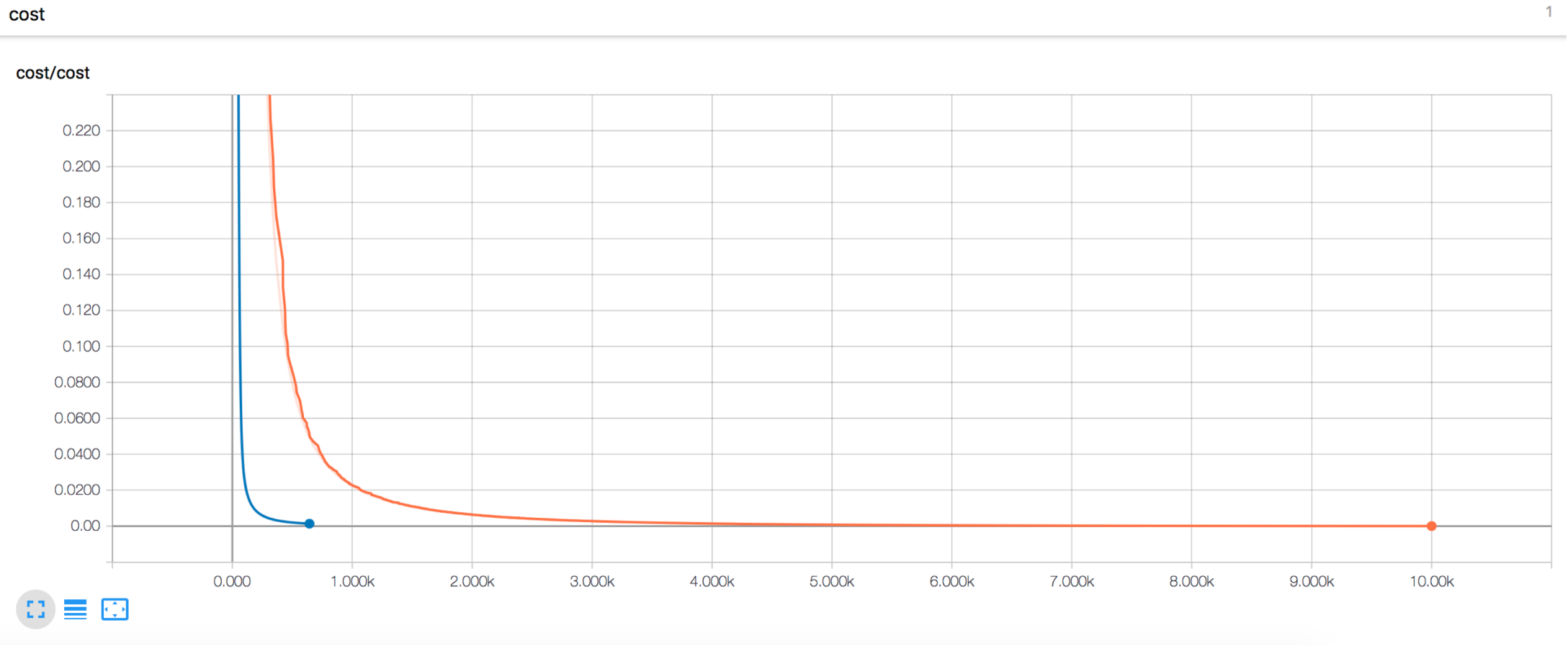
```
W1 = tf.Variable(tf.random_normal([2, 10]), name='weight1')
b1 = tf.Variable(tf.random_normal([10]), name='bias1')
layer1 = tf.sigmoid(tf.matmul(X, W1) + b1)

W2 = tf.Variable(tf.random_normal([10, 10]), name='weight2')
b2 = tf.Variable(tf.random_normal([10]), name='bias2')
layer2 = tf.sigmoid(tf.matmul(layer1, W2) + b2)

W3 = tf.Variable(tf.random_normal([10, 10]), name='weight3')
b3 = tf.Variable(tf.random_normal([10]), name='bias3')
layer3 = tf.sigmoid(tf.matmul(layer2, W3) + b3)

W4 = tf.Variable(tf.random_normal([10, 1]), name='weight4')
b4 = tf.Variable(tf.random_normal([1]), name='bias4')
hypothesis = tf.sigmoid(tf.matmul(layer3, W4) + b4)
```

Lab 9-2

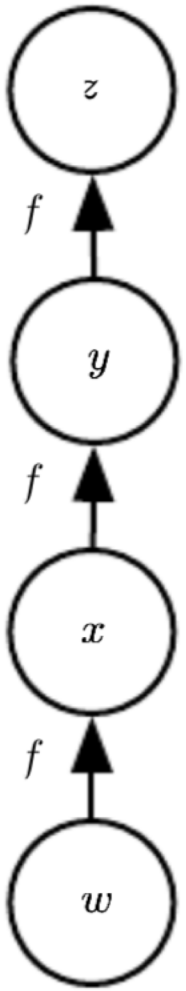
- About Tensorboard
- 자세한 문법은 생략...

Lab 9-2

- 파란색 – Learning Rate 0.1, 주황색 – Learning Rate 0.01



More about **backpropagation**



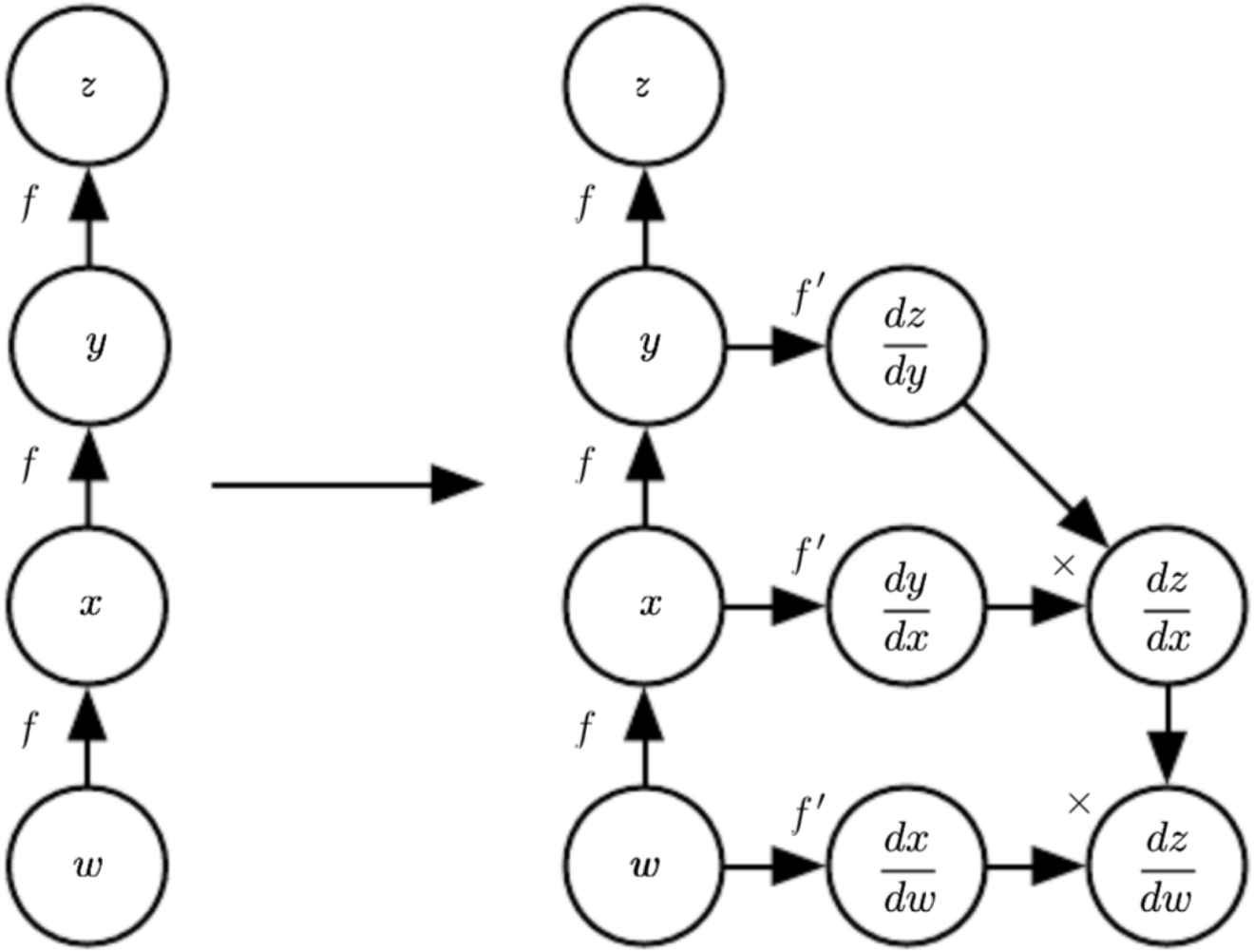
$$\frac{\partial z}{\partial w} \tag{6.50}$$

$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \tag{6.51}$$

$$= f'(y)f'(x)f'(w) \tag{6.52}$$

$$= f'(f(f(w)))f'(f(w))f'(w) \tag{6.53}$$

More about **backpropagation**



More about **backpropagation**

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
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

More about **backpropagation**

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:

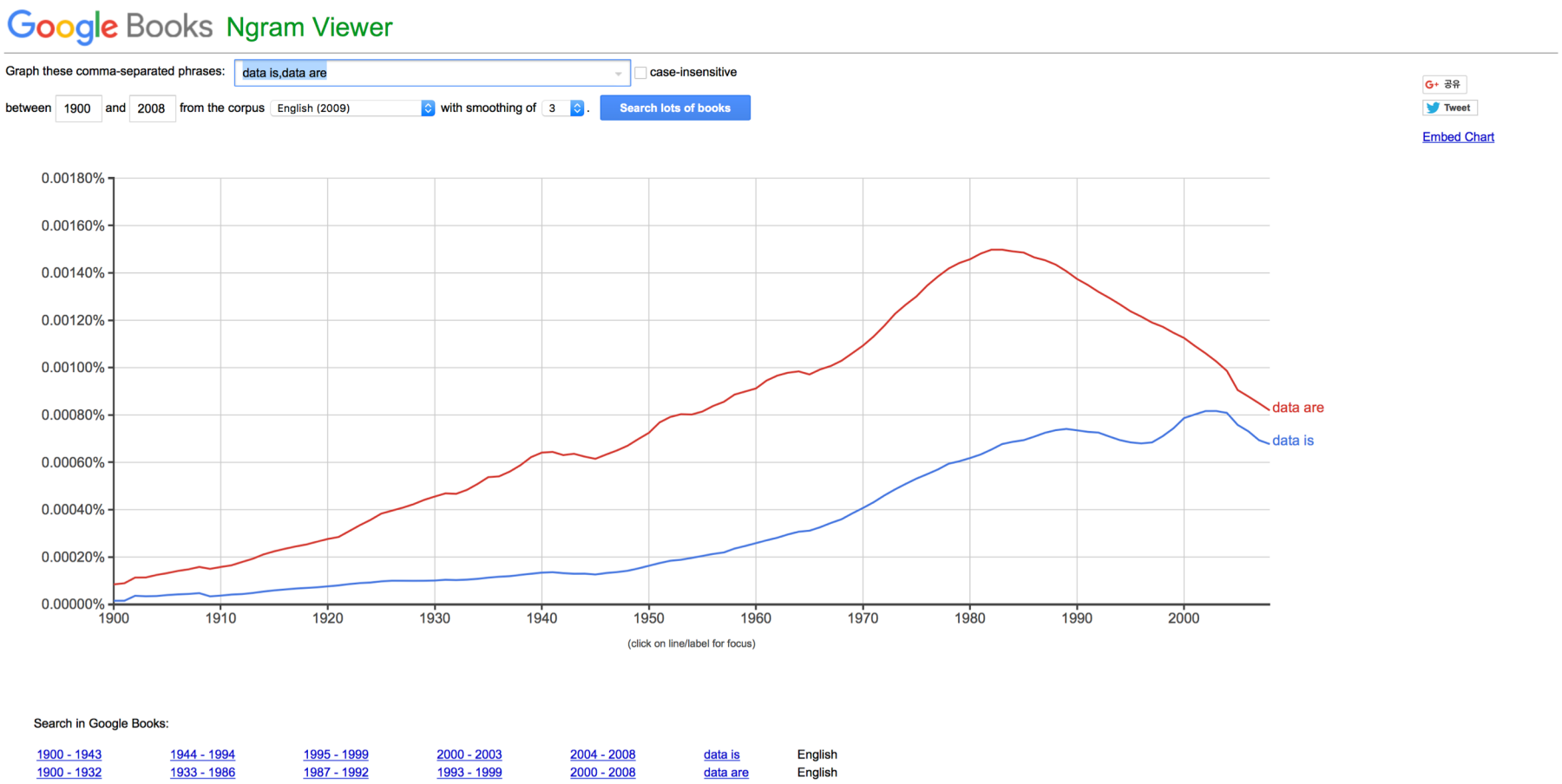
$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

end for

More about **backpropagation**

- 칠판

Google n-gram viewer



https://zariski.wordpress.com/2010/12/22/거대한-코퍼스로-놀기/

How not to build a minimum viable product

How to build a minimum viable product