13&14기 텍스트 세미나

ToBig's 13기 이재빈

Backpropagation and Computation Graph

Contents

Unit 01 | Matrix Gradients for Neural Nets

1. Derivative wrt a weight matrix

$$\frac{\partial s}{\partial \boldsymbol{W}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$$

scalar

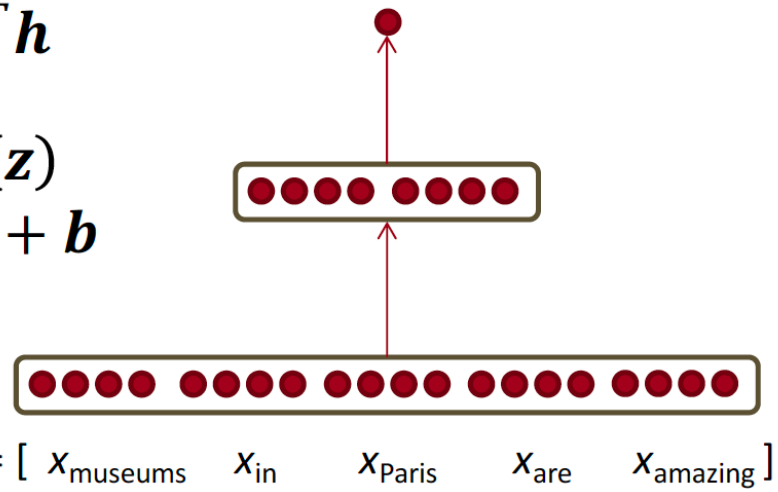
Weight matrix

Chain Rule
함수의 연쇄법칙

합성함수 미분

$$F = (f \circ g)(x) = f\big(g(x)\big)$$
$$F' = (f \circ g)'(x) = f'\big(g(x)\big)g'(x)$$

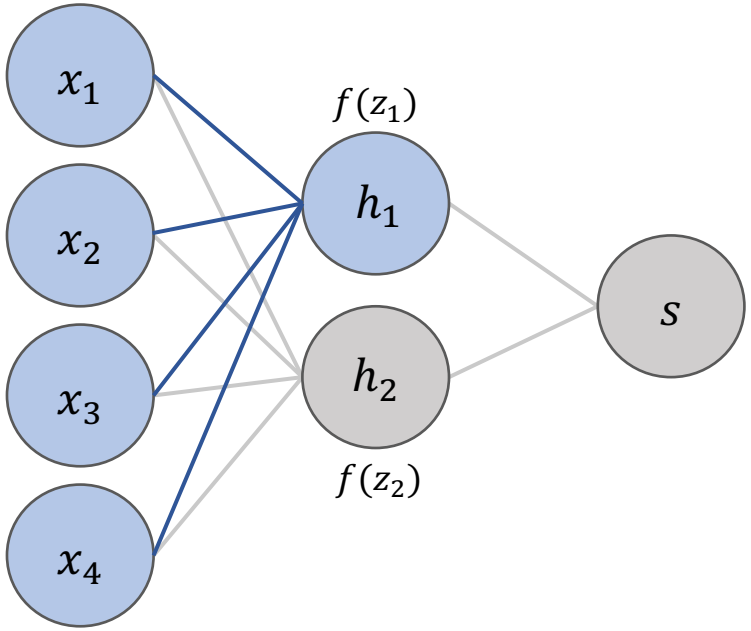
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$



NER
Named Entity Recognition

Unit 01 | Matrix Gradients for Neural Nets

1. Derivative wrt a weight matrix



$$z = Wx$$

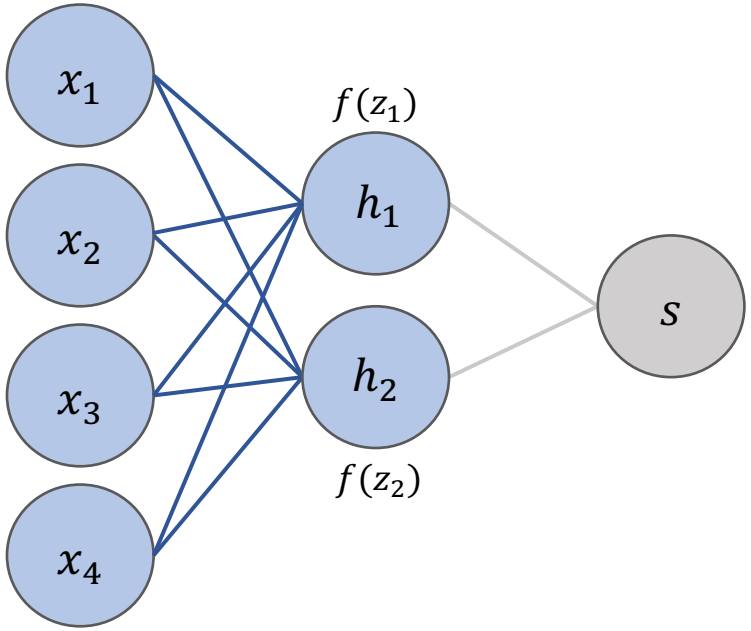
$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

$$z_1 = \sum_{k=1}^{4} w_{1k} x_k$$

$$\rightarrow \quad \frac{\partial z_1}{\partial w_{1j}} = \frac{\partial \sum_{k=1}^{4} w_{1k} x_k}{\partial w_{1j}} = x_j$$

Unit 01 | Matrix Gradients for Neural Nets

1. Derivative wrt a weight matrix



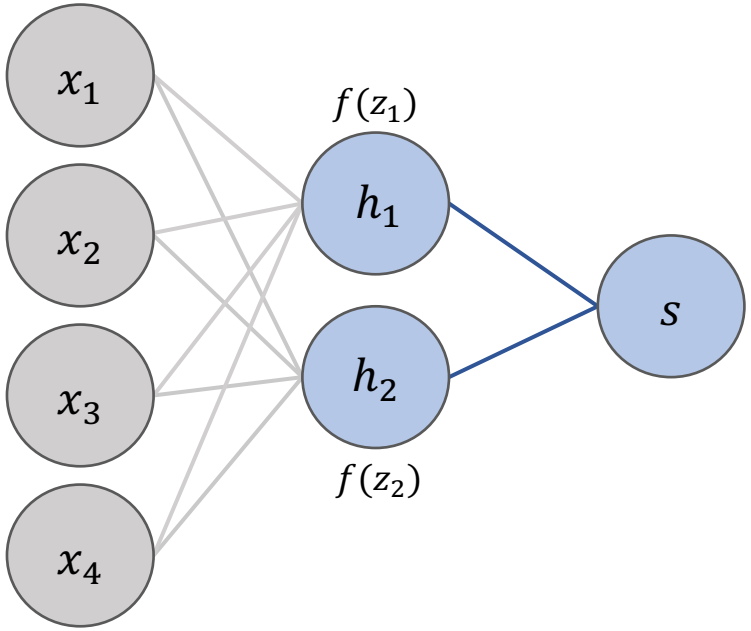
$$z = Wx$$

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

$$\rightarrow \quad \frac{\partial z_i}{\partial w_{ij}} = \frac{\partial \sum_{k=1}^{4} w_{ik} x_k}{\partial w_{ij}} = x_j$$

Unit 01 | Matrix Gradients for Neural Nets

1. Derivative wrt a weight matrix



$$z = Wx$$
$$s = u^T z = u_1 z_1 + u_2 z_2$$

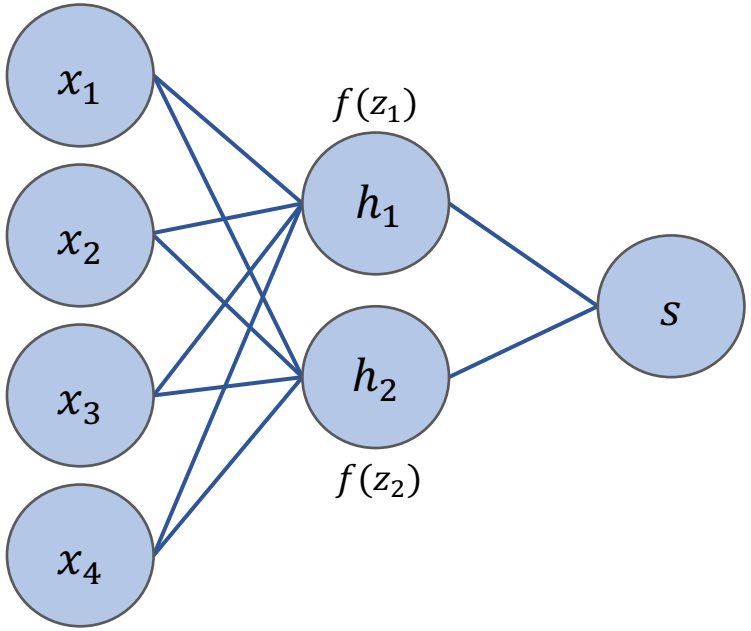
$$\frac{\partial s}{\partial \boldsymbol{W}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} \rightarrow \delta = \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix}$$

$$\frac{\partial s}{\partial W_{ij}} = \delta \frac{\partial z}{\partial W_{ij}} = \sum_{k=1}^{4} \delta \frac{\partial z_k}{\partial W_{ij}} = \delta_i x_j$$

δ : Error signal from above
x : Local gradient signal

Unit 01 | Matrix Gradients for Neural Nets

1. Derivative wrt a weight matrix



$$\frac{\partial s}{\partial W} = \begin{bmatrix}\delta_1 \\ \delta_2\end{bmatrix}[x_1\ x_2\ x_3\ x_4] = \delta x^T$$

m x n n x 1 1 x m

2. Deriving Gradients : Tips

1. 변수를 잘 정의하고 차원을 계속 생각하고 있을 것
2. Chain Rule을 숙지할 것
3. Model의 마지막에서 softmax 계산할 때 correct class / incorrect class 따로 계산
4. 행렬 미분이 헷갈리면 성분 별 미분을 연습하도록..
5. Shape Convention!
   hidden layer에 도착하는 δ의 차원 = 해당 hidden layer의 차원

2. Deriving Gradients : Tips

Model의 마지막에서 softmax 계산할 때 correct class / incorrect class 따로 계산

$$Cross\ Entropy = J = -\sum y_j \log(p_j) = -\sum y_j \log\left(S_j(z)\right)$$

$$\frac{\partial J}{\partial z_i} = -\sum_j \left\{ y_j \frac{\partial}{\partial p_j} \log(p_j) \frac{\partial p_j}{\partial z_i} \right\}$$

$$= -\sum_j \left\{ \frac{y_j}{p_j} \frac{\partial p_j}{\partial z_i} \right\}$$

$$= -y_i(1 - p_i) + \sum_{i \neq j} y_j p_i$$

$$= p_i - y_i (= -(y_i - p_i)$$

$$\frac{\partial p_i}{\partial z_i} = \frac{e^{z_i} \sum e^{z_k} - e^{z_i} e^{z_i}}{(\sum e^{z_k})^2}$$

$$= \frac{e^{z_i}}{\sum e^{z_k}} \left(\frac{\sum e^{z_k} - e^{z_i}}{\sum e^{z_k}} \right)$$

$$= p_i(1 - p_i)$$

$$\frac{\partial p_j}{\partial z_i} = -\frac{e^{z_j}}{\sum e^{z_k}} \frac{e^{z_i}}{\sum e^{z_k}}, \qquad (i \neq j)$$

$$= -p_i p_j$$

* $\boldsymbol{y} = (y_1, \dots, y_k, \dots) = (0,0, \dots, 1, \dots 0)$,
$\boldsymbol{p} = (p_1, \dots, p_k, \dots) = \left(S_1(z_1), S_2(z_2), \dots, S_k(z_k), \dots S_K(z_K)\right)$

$$S(z) = \left(\frac{e^{z_1}}{\sum_{k=1}^{K} e^{z_k}}, \frac{e^{z_2}}{\sum_{k=1}^{K} e^{z_k}}, \dots, \frac{e^{z_K}}{\sum_{k=1}^{K} e^{z_k}} \right)$$

$$p_i = S_i(z) = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}, \qquad \sum S_i(z) = 1$$

$$* \frac{\partial J}{\partial \boldsymbol{z}} = \begin{pmatrix} p_1 - y_1 \\ \vdots \\ p_k - y_k \end{pmatrix}$$

3. Deriving gradient wrt words

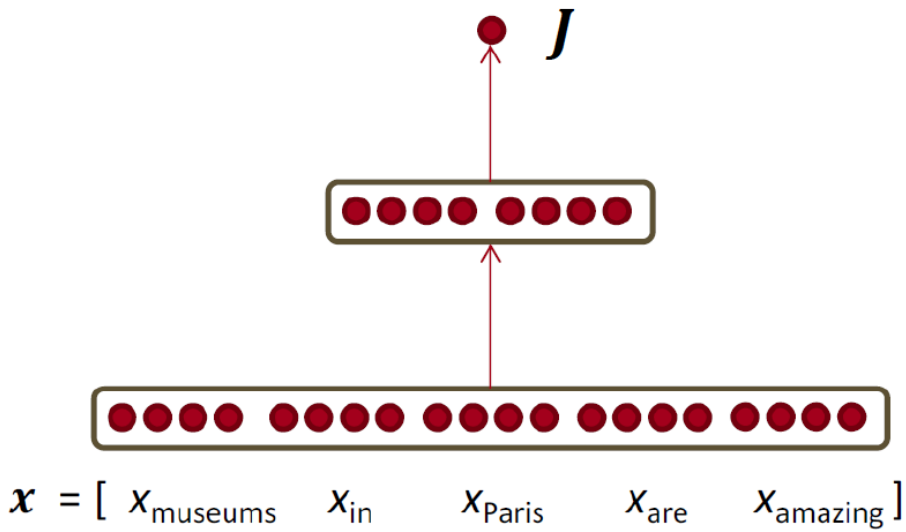
$$\nabla_x J = W^T \delta = \delta_{x_{window}}$$

지금까지는 : W matrix에 대한 미분 확인

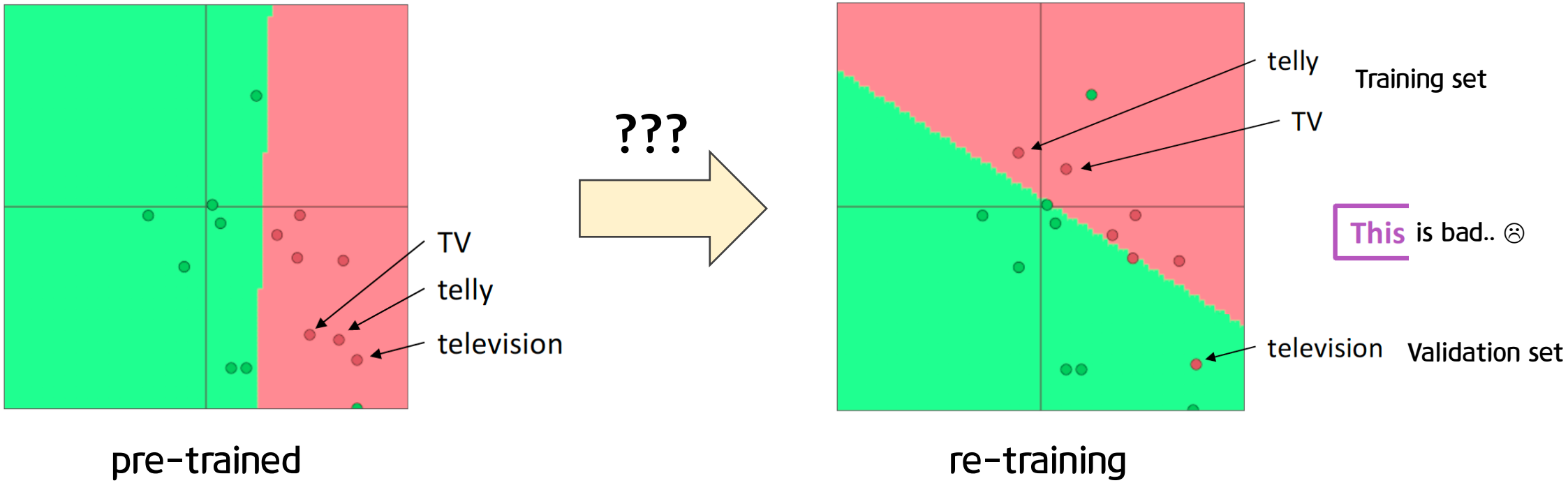
지금부터는 : x 각각의 window 값이 어떻게 gradient 받는지!

Window에 등장한 단어들이 update 되고,

단어벡터들이 task에 더 도움이 되게끔 변화



4. A pitfall when retraining word vectors



4. A pitfall when retraining word vectors

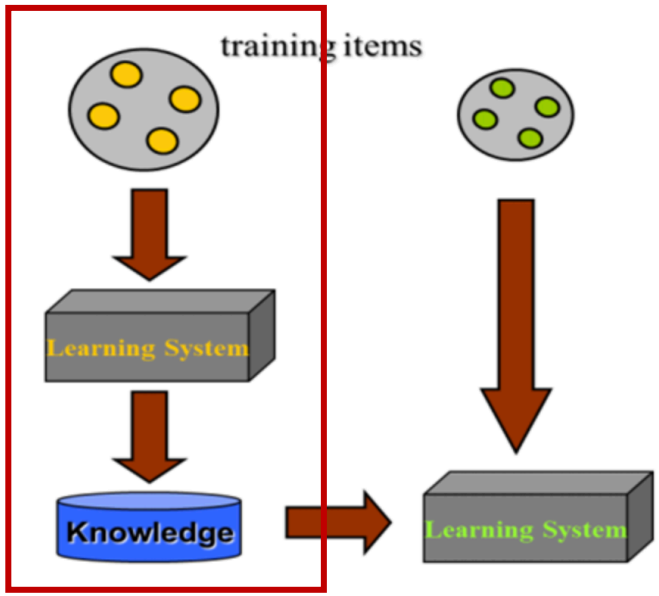
pre-trained vs retraining vs fine-tuning (?)

- pre-trained : 이미 사전에 학습되어 있는 것을 가져다가 쓰는 것
- fine-tuning : pretrained 된 모델을 활용해, task에 맞게 조금 고쳐서 학습시키는 것

- retraining : data를 새로 가져와서, train 과정부터 다시 학습시키는 것

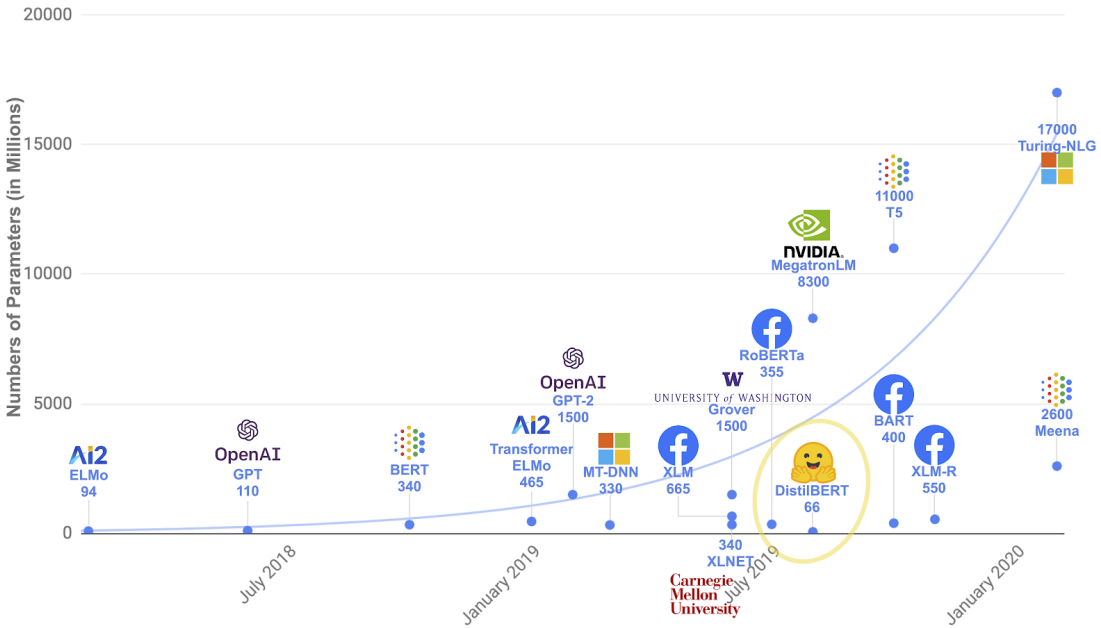
4. A pitfall when retraining word vectors

pre-trained vs retraining vs fine-tuning



pre-trained

- **Multi-Purpose NLP Models**
  - ULMFiT
  - Transformer
  - Google's BERT
  - Transformer-XL
  - OpenAI's GPT-2
- **Word Embeddings**
  - ELMo
  - Flair
- **Other Pretrained Models**
  - StanfordNLP

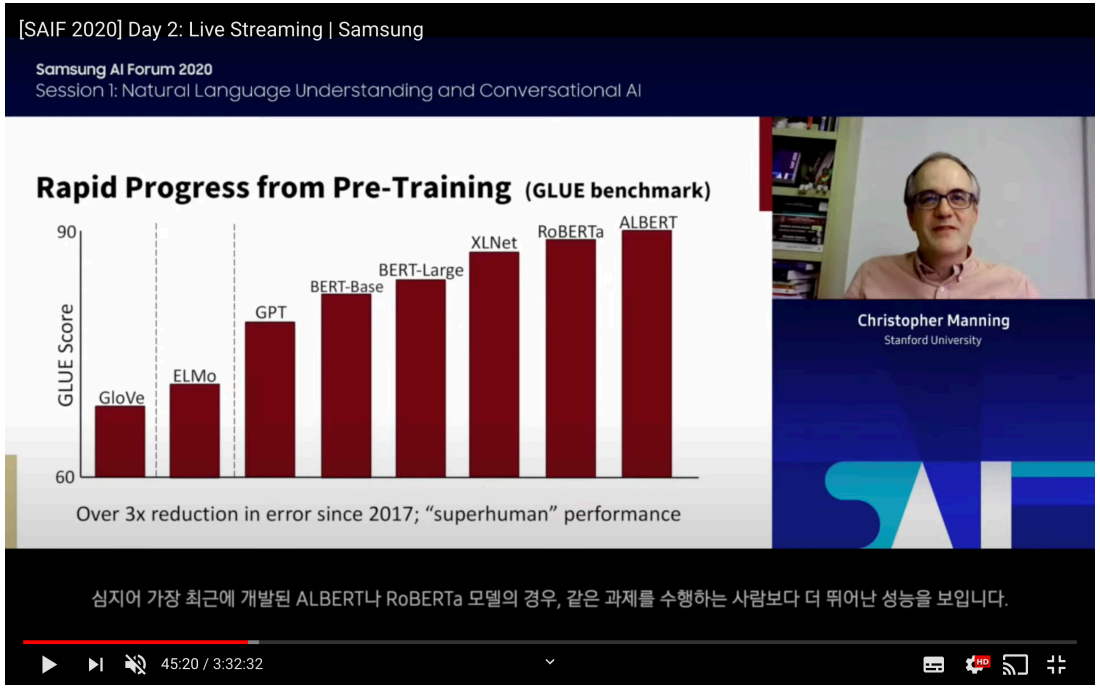


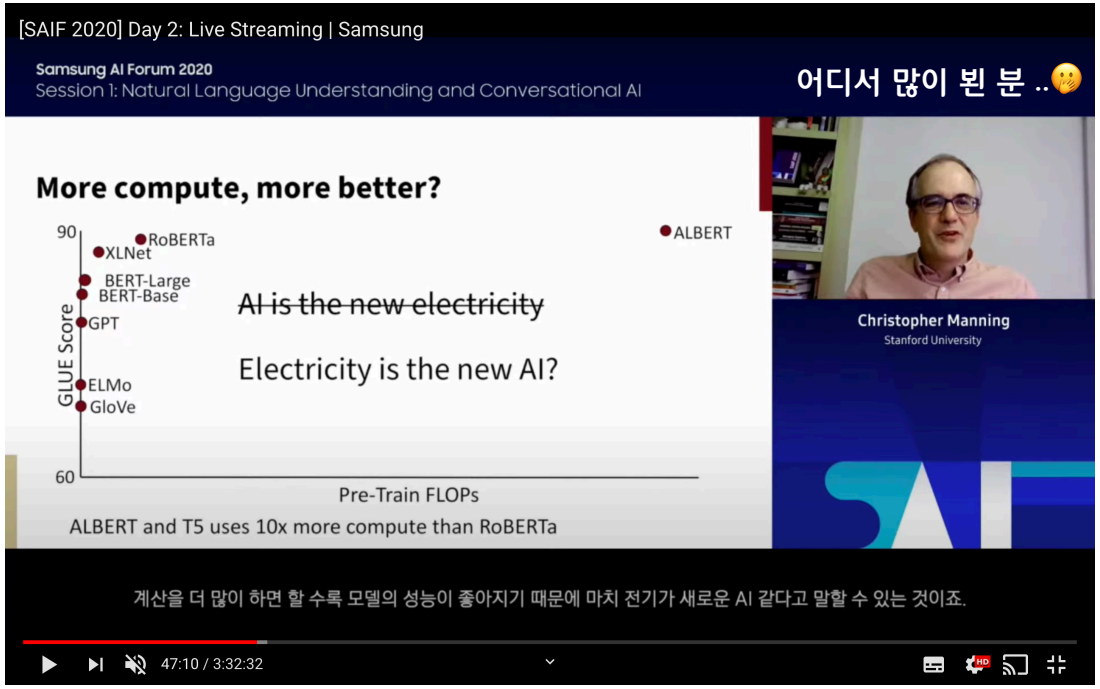
Unit 01 | Matrix Gradients for Neural Nets

https://youtu.be/akgybTjRjNY

4. A pitfall when retraining word vectors

pre-trained vs retraining vs fine-tuning



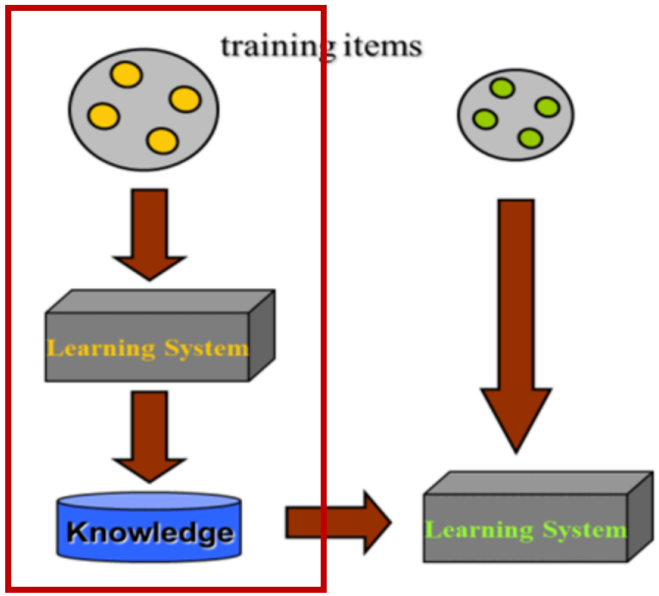


Unit 01 | Matrix Gradients for Neural Nets

http://freesearch.pe.kr/archives/4963

4. A pitfall when retraining word vectors

pre-trained vs retraining vs fine-tuning



pre-trained

```python
# 적절한 BERT model 로딩
bert_base, vocabulary = nlp.model.get_model('bert_12_768_12',
                                            dataset_name='wiki_multilingual_cased',
                                            pretrained=True, ctx=ctx, use_pooler=Tr
                                            use_decoder=False, use_classifier=False
```

Python

```python
ds = gluon.data.SimpleDataset([['나 보기가 역겨워', '김소월']])
tok = nlp.data.BERTTokenizer(vocab=vocabulary, lower=False)
trans = nlp.data.BERTSentenceTransform(tok, max_seq_length=10)

# ds.transform(trans)
```

Python

```python
class BERTDataset(Dataset):
    def __init__(self, dataset, sent_idx, label_idx, bert_tokenizer, max_len,
                 pad, pair):
        transform = nlp.data.BERTSentenceTransform(
            bert_tokenizer, max_seq_length=max_len, pad=pad, pair=pair)
        sent_dataset = gluon.data.SimpleDataset([[
            i[sent_idx],
        ] for i in dataset])
        self.sentences = sent_dataset.transform(transform)
        self.labels = gluon.data.SimpleDataset(
            [np.array(np.int32(i[label_idx])) for i in dataset])

    def __getitem__(self, i):
        return (self.sentences[i] + (self.labels[i], ))

    def __len__(self):
        return (len(self.labels))
```

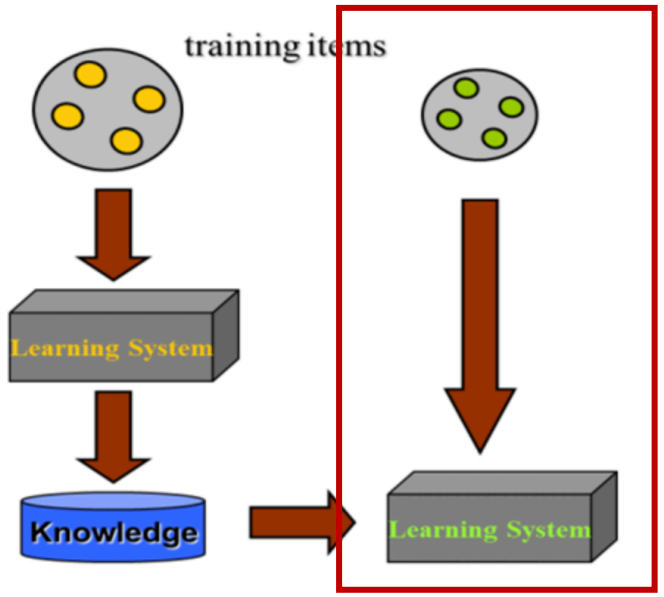
Python

Unit 01 | Matrix Gradients for Neural Nets

http://freesearch.pe.kr/archives/4963

4. A pitfall when retraining word vectors

pre-trained vs retraining vs fine-tuning



fine-tuning

```python
# Fine-Tuning
class BERTClassifier(nn.Block):
    def __init__(self,
                 bert,
                 num_classes=2,
                 dropout=None,
                 prefix=None,
                 params=None):
        super(BERTClassifier, self).__init__(prefix=prefix, params=params)
        self.bert = bert
        with self.name_scope():
            self.classifier = nn.HybridSequential(prefix=prefix)
            if dropout:
                self.classifier.add(nn.Dropout(rate=dropout))
            self.classifier.add(nn.Dense(units=num_classes))

    # pooler 결과를 받아, FC layer 추가
    def forward(self, inputs, token_types, valid_length=None):
        _, pooler_out = self.bert(inputs, token_types, valid_length)
        return self.classifier(pooler_out)
```

Python

4. A pitfall when retraining word vectors

딴 길로 좀 샜는데 다시 본론으로 돌아오면.. ☺

1. **웬만하면 pre-trained word vector를 이용할 것**
   - Pre-trained data는 방대한 양에 대해 사람들이 이미 학습을 시켜 놓았음
   - TV, telly, television 처럼 훈련 집합 포함의 유무와 관계 없이 어느 정도 단어 간 관계 형성!
   - 그러나 데이터가 매우 많다면 (1억 개 이상), 처음부터 학습을 시켜도 무관함

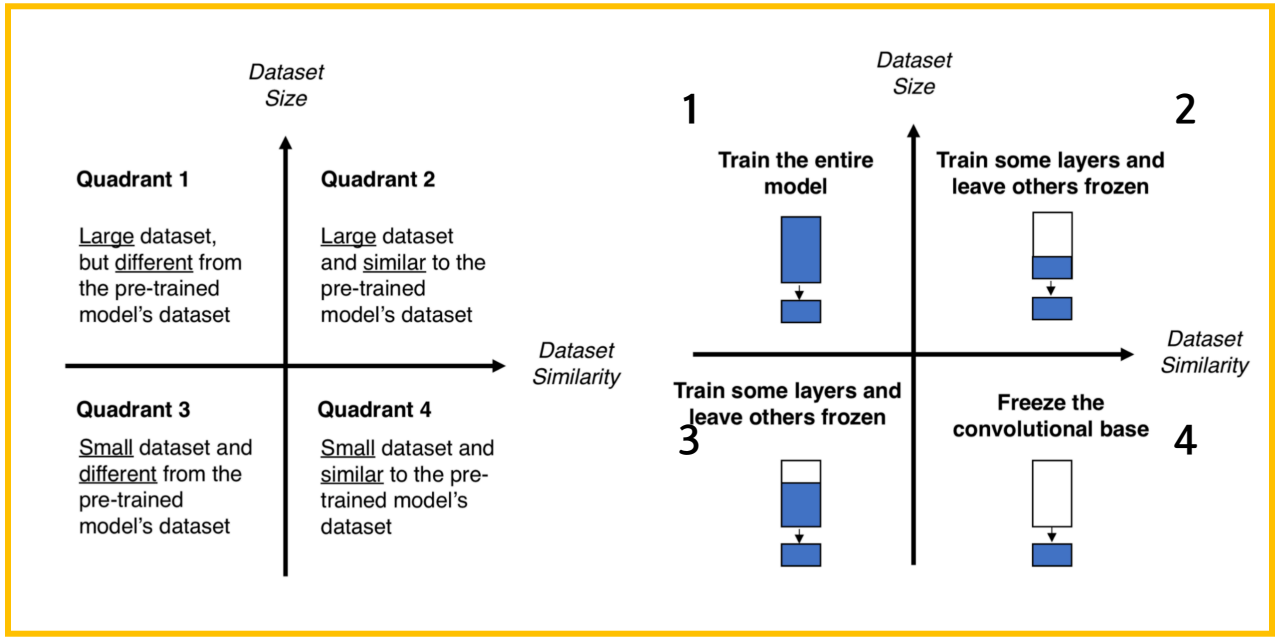
2. **Retraining word vector (=fine tuning) 을 해야 하는 경우**
   - 만약 적은 training dataset(100,000개 미만)을 가지고 있다면 fine-tuning 하지 마시오
   - 많은 training set(100만 개 이상)을 가지고 있다면 fine-tuning 이 성능에 도움이 될 것

https://jeinalog.tistory.com/13

4. A pitfall when retraining word vectors

Fine-Tuning 진행 방법 결정

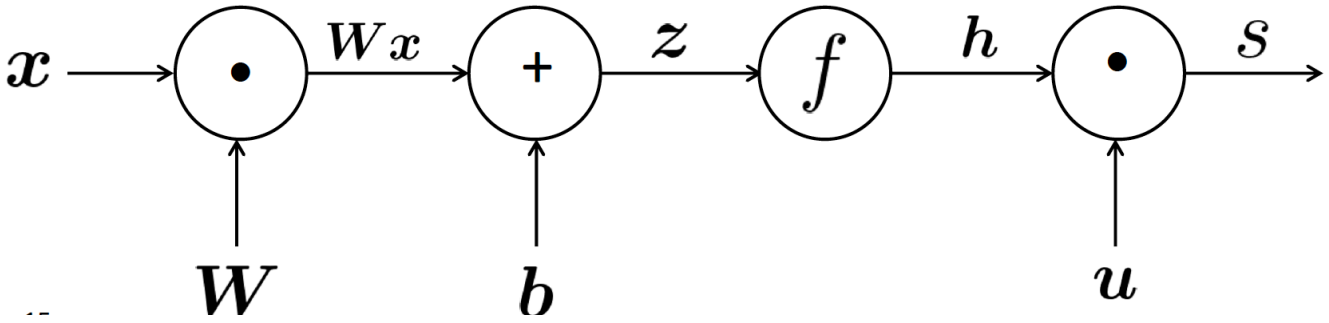
1. **크기가 크고 유사성이 낮은 데이터**
   - 데이터가 크므로, 내가 원하는 대로 다시 학습
   - Pre-trained 되어 있는 것으로 하면 무에서 유보다는 좀 더 유용할 듯
2. **크기가 크고 유사성도 높은 데이터 (good)**
   - Classifier와 모델의 마지막 단 계층 일부만 학습시켜도 충분
3. **크기도 작고 유사성도 낮은 데이터 (..)**
   - 너무 많은 계층을 새로 학습시키면 underfitting
   - 너무 적은 계층만을 학습시키면 제대로 학습되지 않을 것..
   - Data Augmentation
4. **크기가 작지만 유사성은 높은 데이터**
   - 새로운 classifier 만 만들어서 학습시킨다

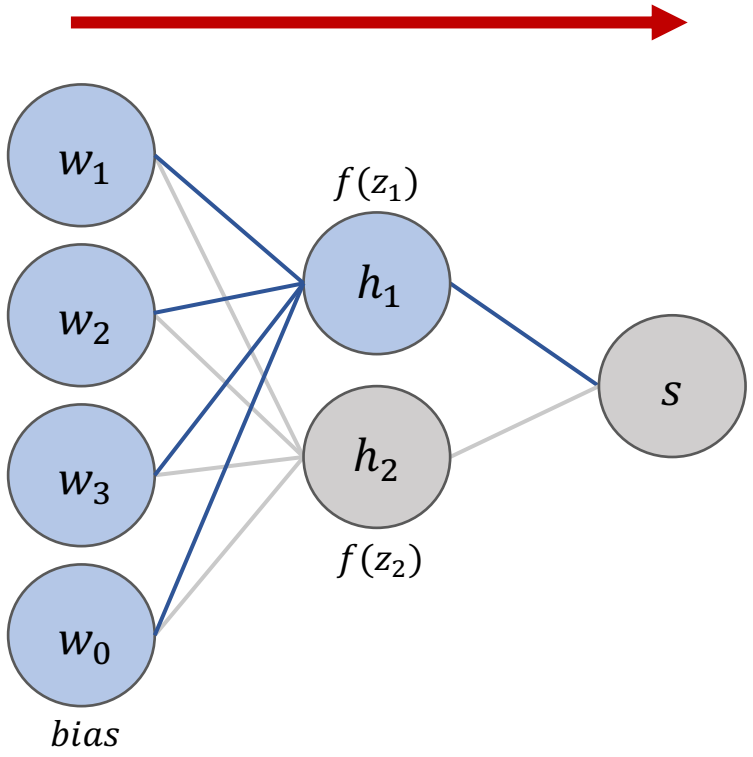


Contents

1. Forward & Back Propagation

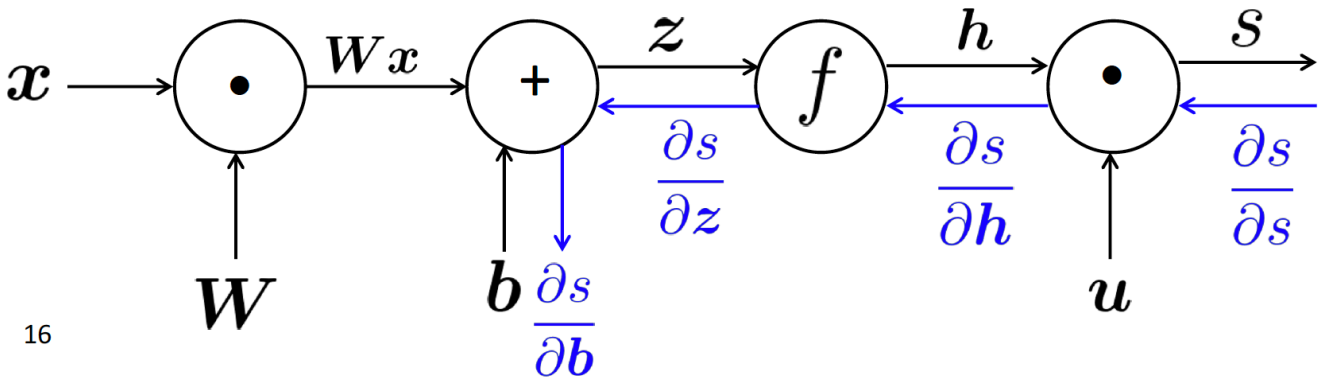
Forward Propagation



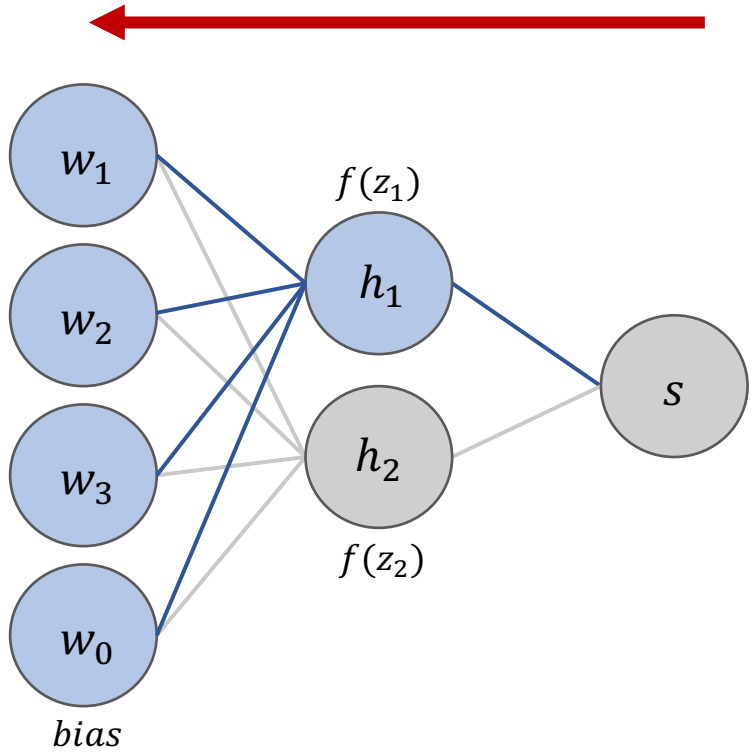


1. Forward & Back Propagation

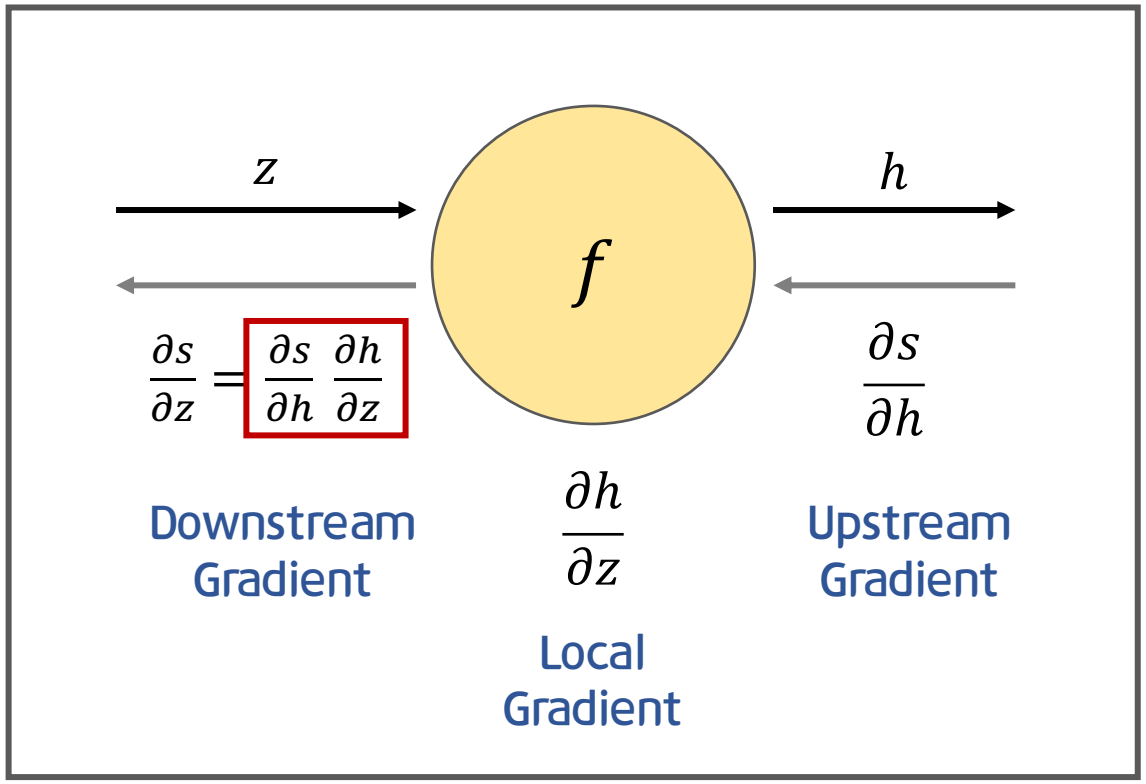
Back Propagation



16



1. Forward & Back Propagation

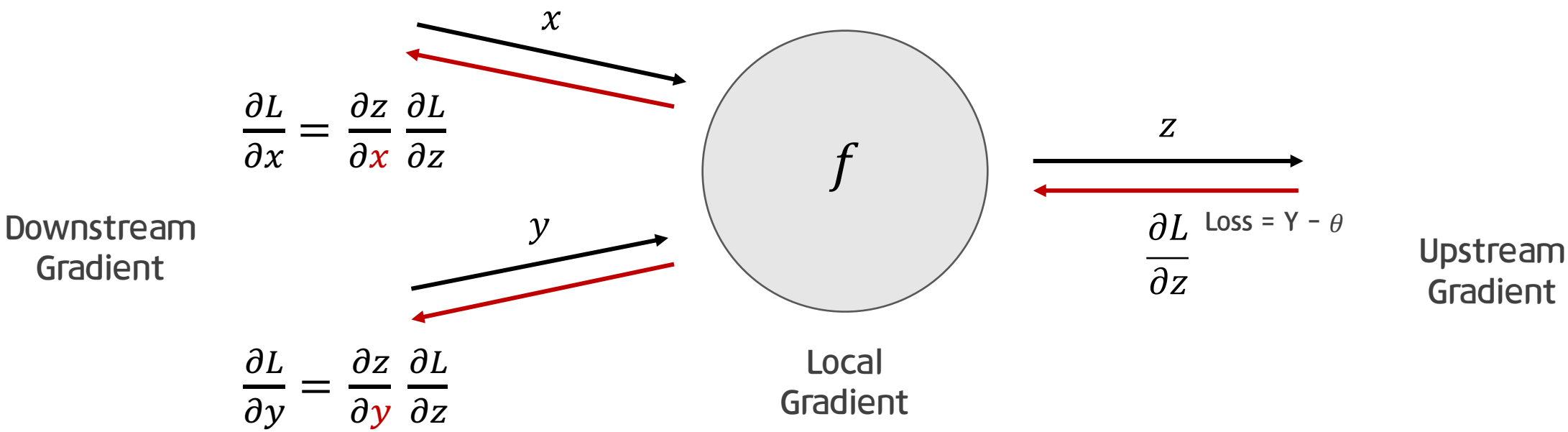


1. Local Gradient = $\frac{\partial(output)}{\partial(input)}$

2. Downstream Gradient
   = Local Gradient * Upstream Gradient
   (∵ Chain Rule)

2. Backpropagation

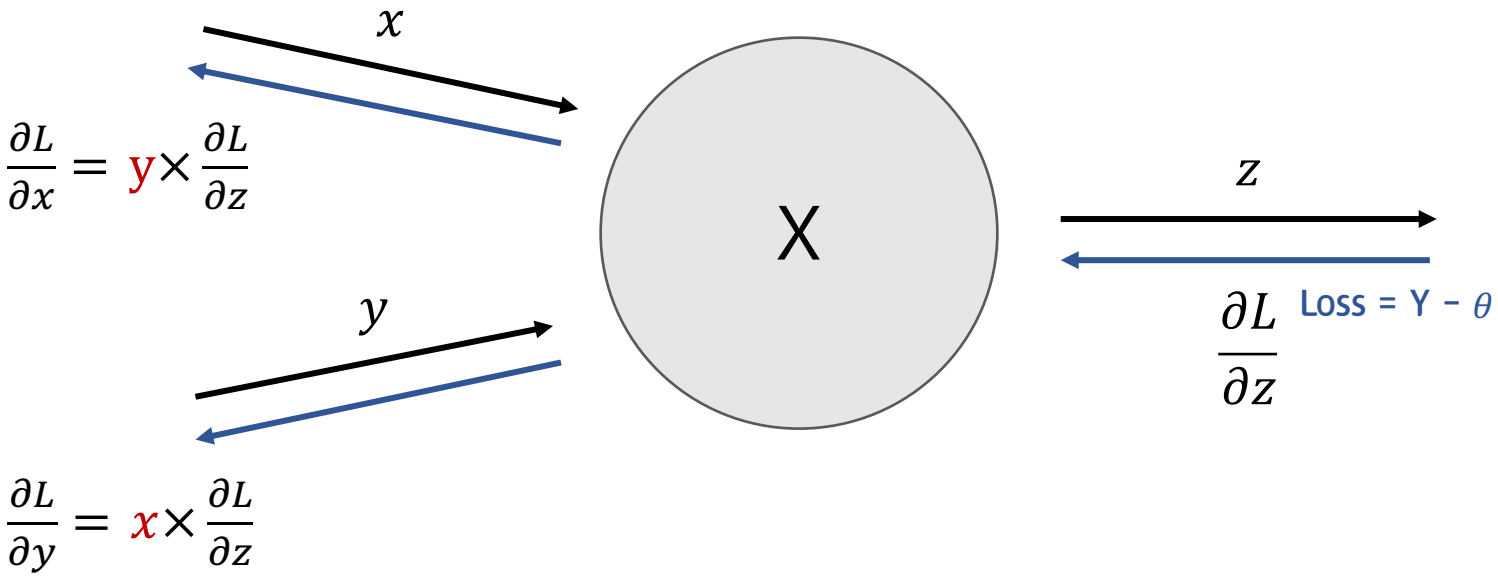
역전파 분해



2. Backpropagation

곱셈의 역전파

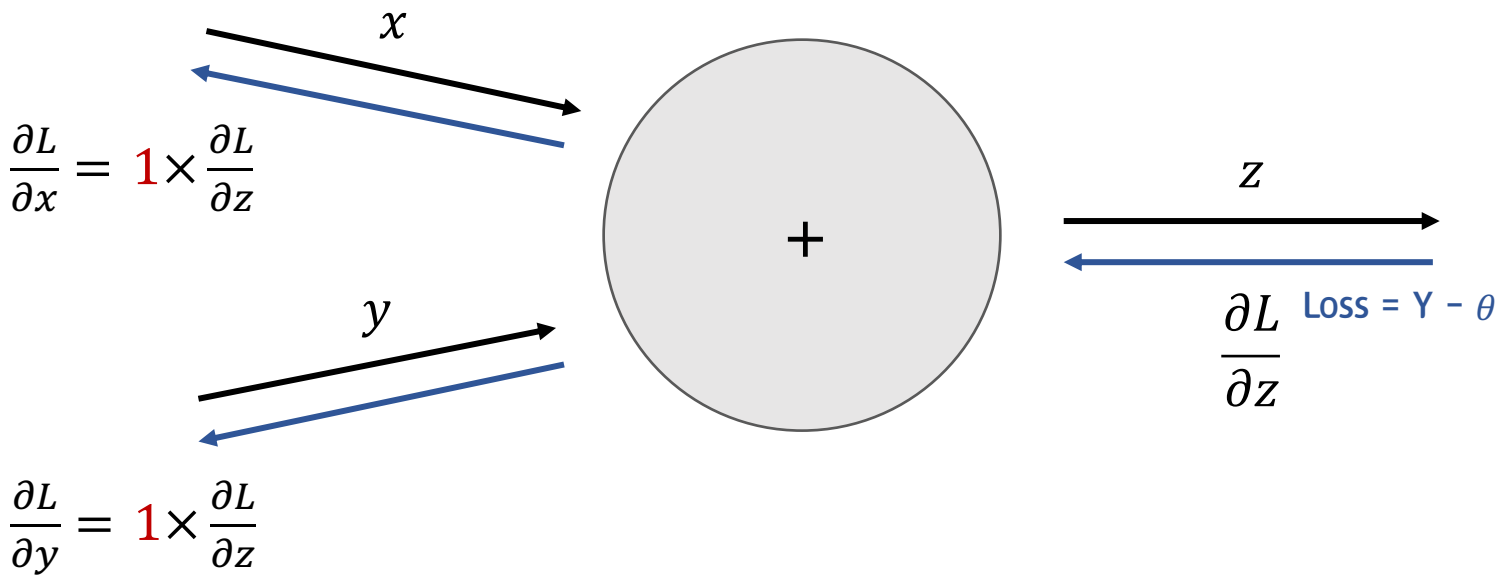
$$z = f(x, y) = xy$$



2. Backpropagation

덧셈의 역전파

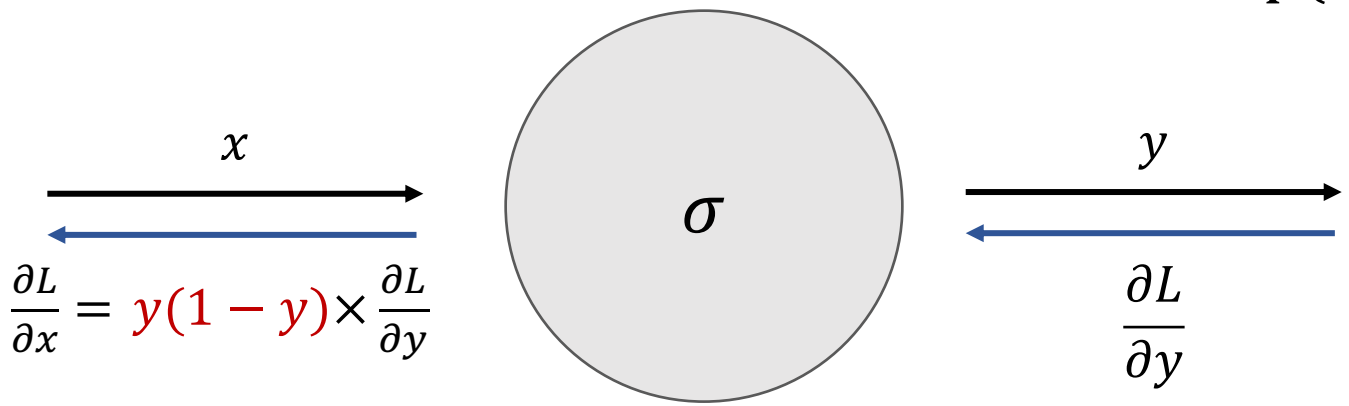
$$z = f(x, y) = x + y$$



2. Backpropagation

시그모이드 역전파

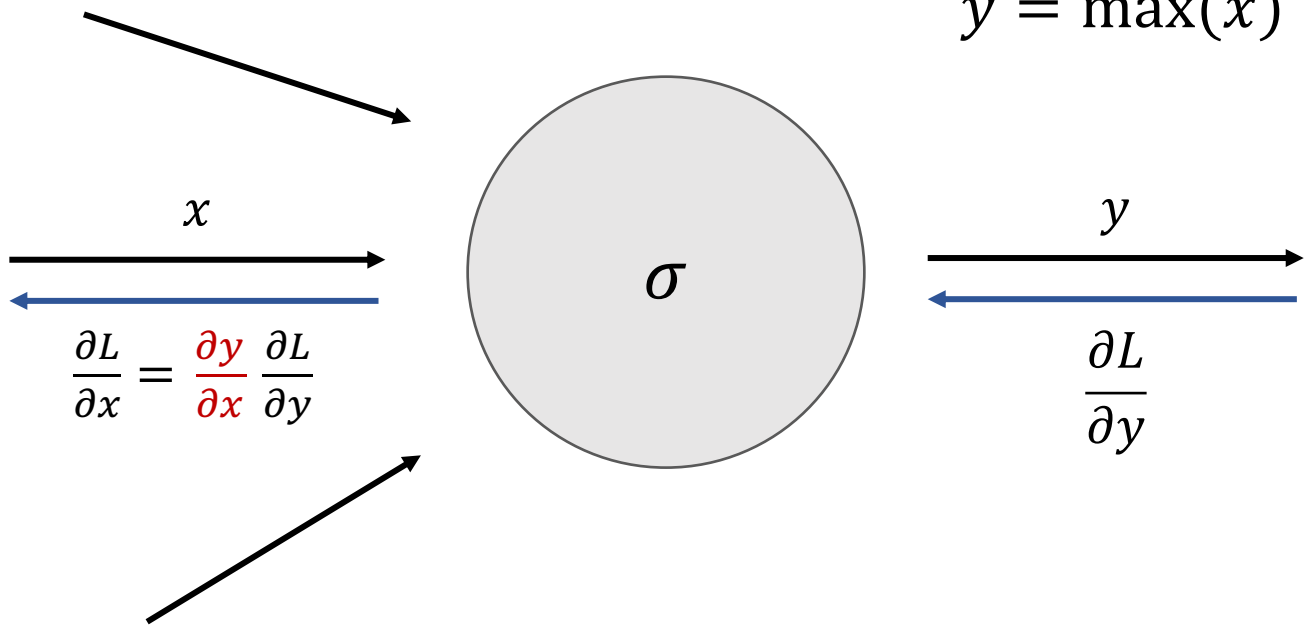
$$y = \frac{1}{1 + \exp(-x)}$$



2. Backpropagation

max 역전파

$$y = \max(x)$$

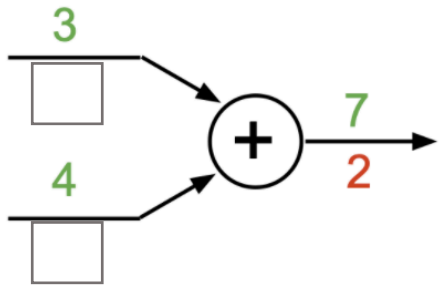


$$\frac{\partial y}{\partial x} = \begin{cases} 1 \; if \; x \; is \; max \\ 0 \; otherwise \end{cases}$$

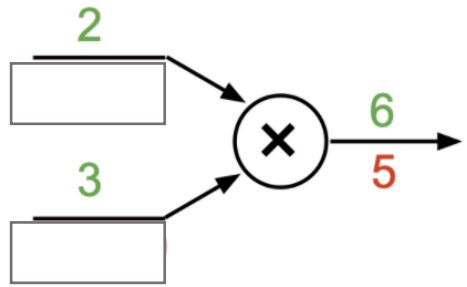
2. Backpropagation

Gradient Flow

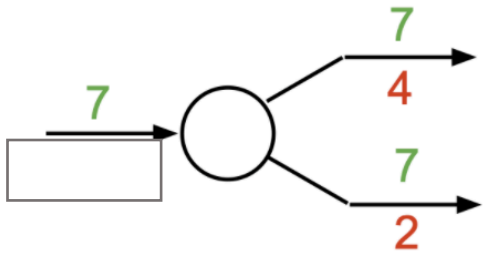
add gate: gradient distributor



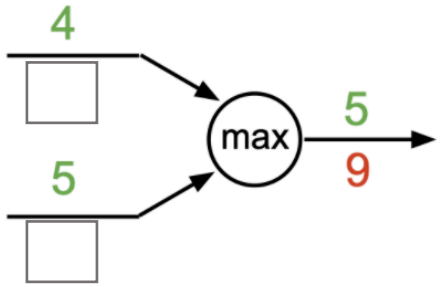
mul gate: "swap multiplier"



copy gate: gradient adder



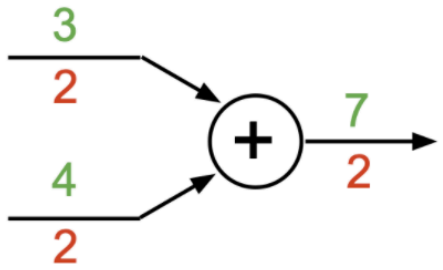
max gate: gradient router



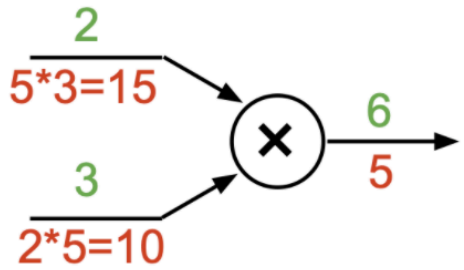
2. Backpropagation

Gradient Flow

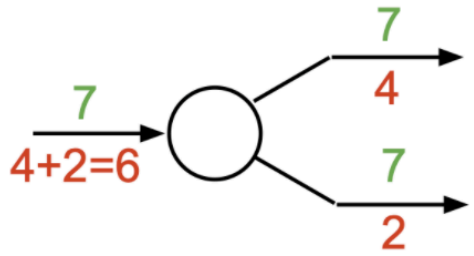
add gate: gradient distributor



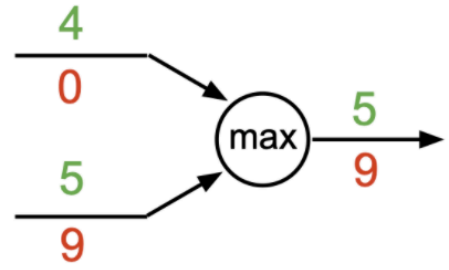
mul gate: "swap multiplier"



copy gate: gradient adder



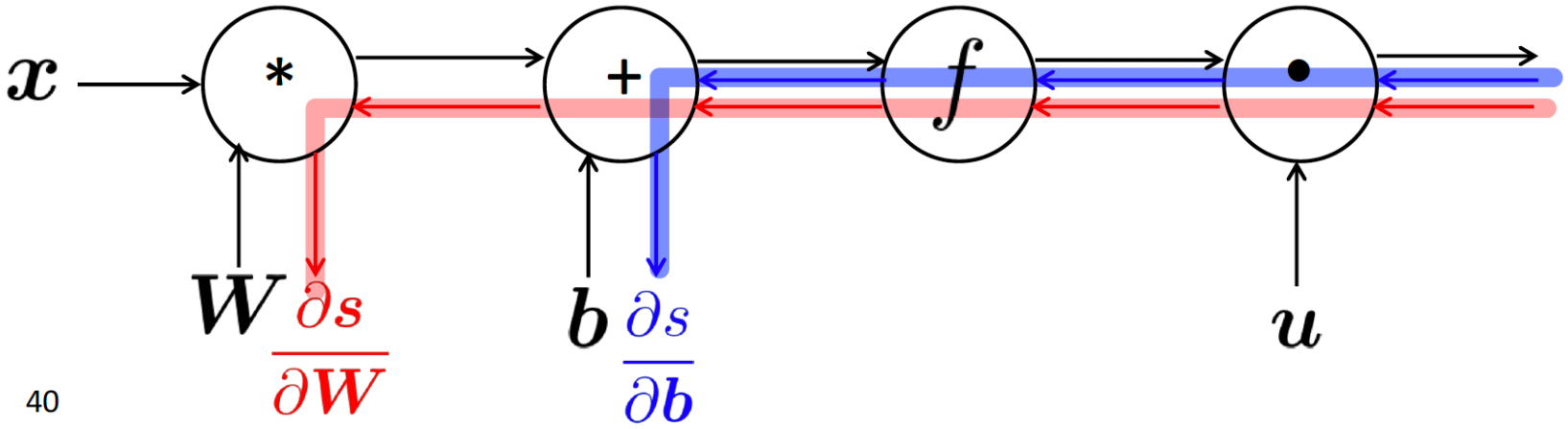
max gate: gradient router



3. Efficiency

Compute all gradients at once

- $\frac{\partial s}{\partial b}$ 를 계산한 다음에 $\frac{\partial s}{\partial W}$ 계산하면 비효율적이다.. 한번에 계산하자 with Chain Rule ~

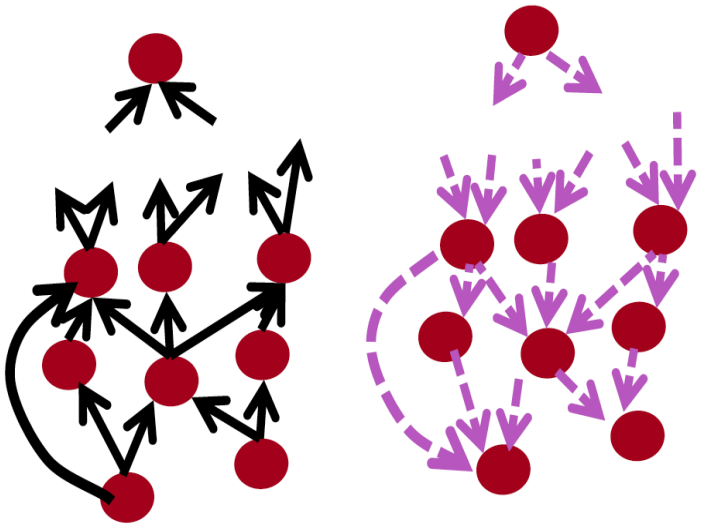


4. Computation Graph

1. Forward : Topological sort 로 정렬한 뒤 노드를 지남
   (사이클이 없는 방향 그래프의 노드들이 방향성을 거스르지 않도록 노드 정렬)

2. Backward : output node 에서 1로 gradient 시작
   Topological sort 의 반대 순서로 노드를 지나며
   이어지는 노드에 대해 local gradient 계산

3. 순전파 과정과 역전파 과정의 시간복잡도는 동일함



Forward Backward

4. 딥러닝 프레임워크 (Pytorch, Tensorflow) backpropagation 너무 잘 구현해 놓음 ^_^

4. Computation Graph

Numerical Gradient

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

- Gradient Check !
- 미분 공식을 통해 Gradient 쉽게 계산 가능
- But 적용할 때 마다 f에 대해 새로 계산해야 하기 때문에, computation graph 방식에 비해 매우 느림
- 따라서 일부에 대해서만 gradient가 제대로 계산되었는지 확인

Unit 02 | Computation Graphs and Backpropagation

https://kasperfred.com/series/introduction-to-neural-networks/computational-complexity-of-neural-networks
http://norman3.github.io/prml/docs/chapter05/3.html

4. Computation Graph

Numerical Gradient

5.3.3 역전파 알고리즘의 효율 (Efficiency of backpropagation)

- 역전파에서 가장 중요한 것 중 하나로 계산의 효율성을 들 수 있다.
- 이 문제를 이해하기 위해서 실제 에러 함수를 미분하여 평가하는 작업이 얼마나 많은 컴퓨터 연산을 발생시키는지 확인해보도록 하자.
  - 이 때 네트워크에서 구해야 하는 가중치 W 의 수를 함께 고려한다.
- 에러 함수를 평가하는 과정에서는 매우 많은 갯수의 파라미터 W 에 대해 $O(W)$ 만큼의 연산이 필요하게 된다.
- 그리고 파라미터의 개수는 유닛의 수보다 크다. (물론 sparse 모델은 이렇지 않을수도 있겠지만 흔한 모델은 아니다.)
  - 따라서 식 (5.48)에 의해 전방향 전파는 $O(W)$ 의 계산 비용이 필요한다.
- 반면 역전파의 경우에는 미분을 위해 유한 차분(finite difference) 기법을 도입해다고 생각하고 이에 대한 비용을 근사적으로 예측해보자.

$$a_j = \sum_i w_{ji} z_i \qquad (5.48)$$

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji} + e) - E_n(w_{ji})}{e} + O(e) \qquad (5.68)$$

- 보통 $e \ll 1$ 이다. 정확도를 향상시키려면 e 는 매우 작은 값이어야 한다.

Forward	Backward
$n_{mul} = n_{layers} \cdot n^3$	$\text{time}_{\text{gradient descent}} = n_{\text{gradient iterations}} \cdot \text{time}_{\text{weights}}$
$n_{mul} = O(n \cdot n^3) = O(n^4)$	$O(\text{time}_{\text{gradient descent}}) = n \cdot n^4 = n^5$

- (참고) **finite difference method**
  - 유한 차분법은 $f(x+b) - f(x+a)$ 형태의 수학 식을 의미한다.
  - 그리고 이를 $(b-a)$ 값으로 나누게 되면 *difference quotient* 를 얻는다.
  - 이 때 $|b-a|$ 가 0에 가까워지면 이를 미분 근사로 생각할 수 있다.
  - 따라서 *finite difference method* 는 수치적 미분 방식을 계산하기 위한 도구이다.
- 수치적인 미분식을 사용하는 경우에는 역전파의 계산 비용이 $O(W)$ 로는 불가능하다.
- 정확도를 높이기 위해 *symmetrical central differences* 를 사용할 수도 있다.

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji} + e) - E_n(w_{ji} - e)}{2e} + O(e^2) \qquad (5.69)$$

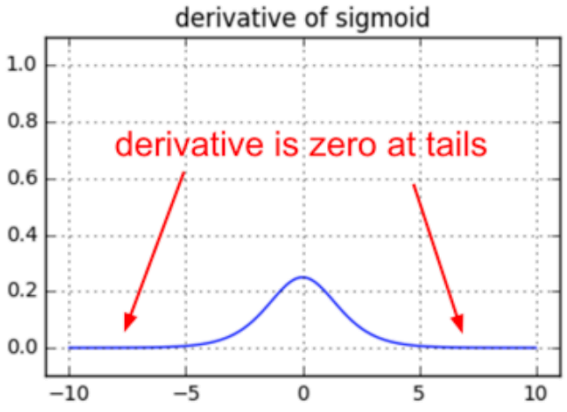
- 참고로 위의 식은 테일러 급수를 이용하여 얻어낸 식이다.
  - 전개를 해 보면 마지막 차항이 $O(e)$ 가 아닌 $O(e^2)$ 으로 나오는지 그 이유를 알 수 있다.
  - 따라서 전방 전파시에는 $O(W)$ 이고 역전파에서는 $O(W^2)$ 의 비용이 소요된다.
- 실제로 이러한 수학적 미분 방식을 통한 계산 식이 매우 중요한데, 다른 방식을 이용하여 연전파를 구한 값과 비교를 할 수 있는 대상으로 사용할 수 있기 때문이다.
  - 따라서 다른 방식으로 얻어지는 결과가 올바른지를 비교해 볼 수 있는 방법으로 응용이 가능하다.

contents

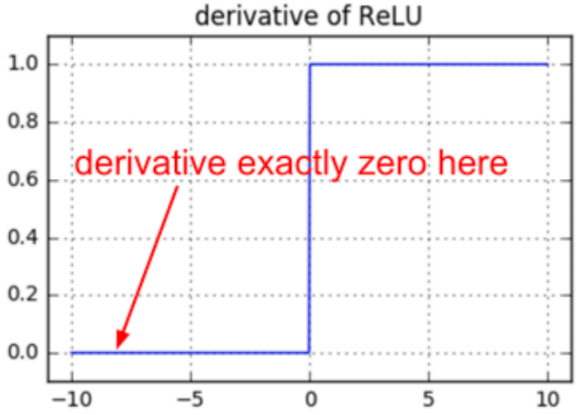
Unit 03 | Tips and Tricks for Neural Networks

https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b

0. Backpropagation 문제..



Vanishing Gradients on Sigmoid



Dying ReLU

```
H = 5    # dimensionality of hidden state
T = 50   # number of time steps
Whh = np.random.randn(H,H)

# forward pass of an RNN (ignoring inputs x)
hs = {}
ss = {}
hs[-1] = np.random.randn(H)
for t in xrange(T):
    ss[t] = np.dot(Whh, hs[t-1])
    hs[t] = np.maximum(0, ss[t])

# backward pass of the RNN
dhs = {}
dss = {}
dhs[T-1] = np.random.randn(H) # start off the chain with random gradient
for t in reversed(xrange(T)):
    dss[t] = (hs[t] > 0) * dhs[t] # backprop through the nonlinearity
    dhs[t-1] = np.dot(Whh.T, dss[t]) # backprop into previous hidden state
```

if the largest eigenvalue is > 1, gradient will explode
if the largest eigenvalue is < 1, gradient will vanish

Exploding Gradients in vanilla RNNs

0. Backpropagation 문제..

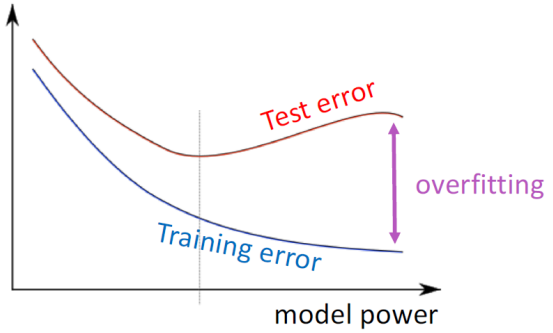
- Vanishing Gradient -> Activation Function, Weight Initialization
- Too slow -> Batch Normalization, Optimization
- Overfitting -> Regularization, Dropout

Unit 03 | Tips and Tricks for Neural Networks

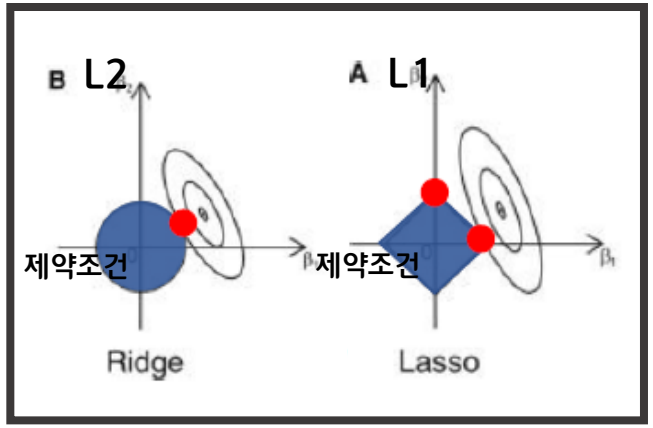
1. Regularization

https://rk1993.tistory.com/entry/Ridge-regression%EC%99%80-Lasso-regression-%EC%89%BD%EA%B2%8C-%EC%9D%B4%ED%95%B4%ED%95%98%EA%B8%B0

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right) + \lambda \sum_k \theta_k^2$$



- biased 하지만 smaller variance를 갖는 estimator
- 모델이 복잡해질수록 penalty를 크게 주도록, 목적 함수에 항을 하나 더 추가



- 보통 L2 Regularization 많이 사용
- 접하는 범위까지 최소제곱의 오차를 허용하겠다 ~

2. Vectorization

```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

Word vector를 하나씩 돌아가며 W와 내적

<

10배
빠름

```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

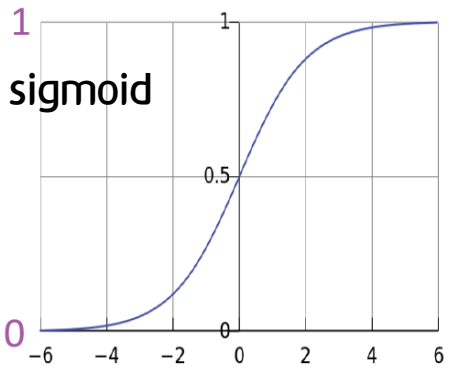
Word vector 집합을 하나의 행렬로 만든 뒤
W와 내적

3. Activation Function

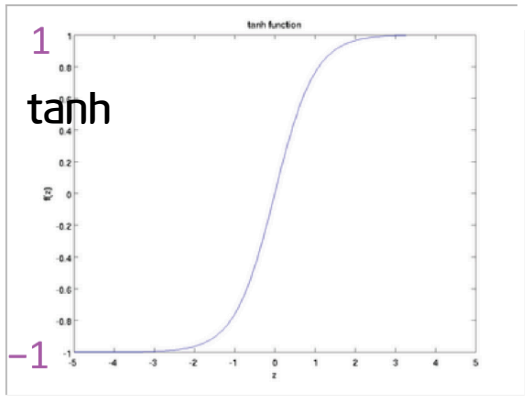
Non-linearity

- tanh 는 sigmoid 를 rescale 한 것
- sigmoid, tanh 는 deep한 구조에서는 잘 사용하지 않는다 -> exp 때문에 연산량 많아져서..

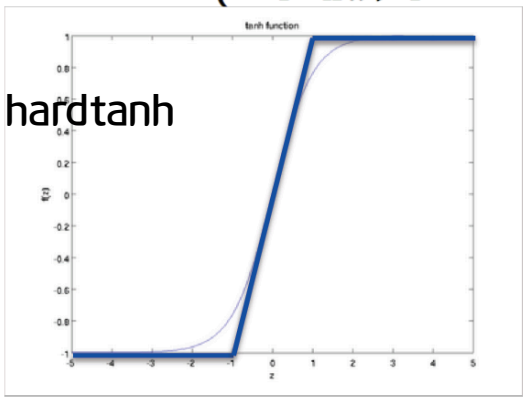
$$f(z) = \frac{1}{1 + \exp(-z)}$$



$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$



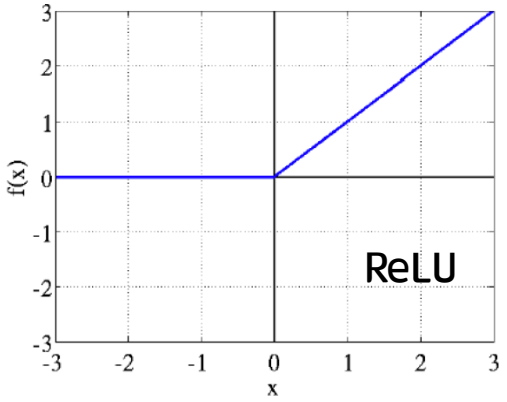
$$\text{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$

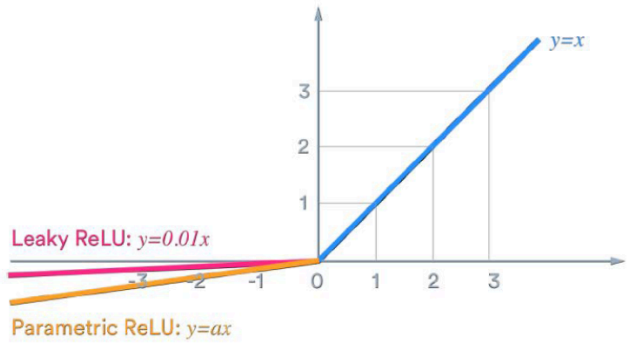


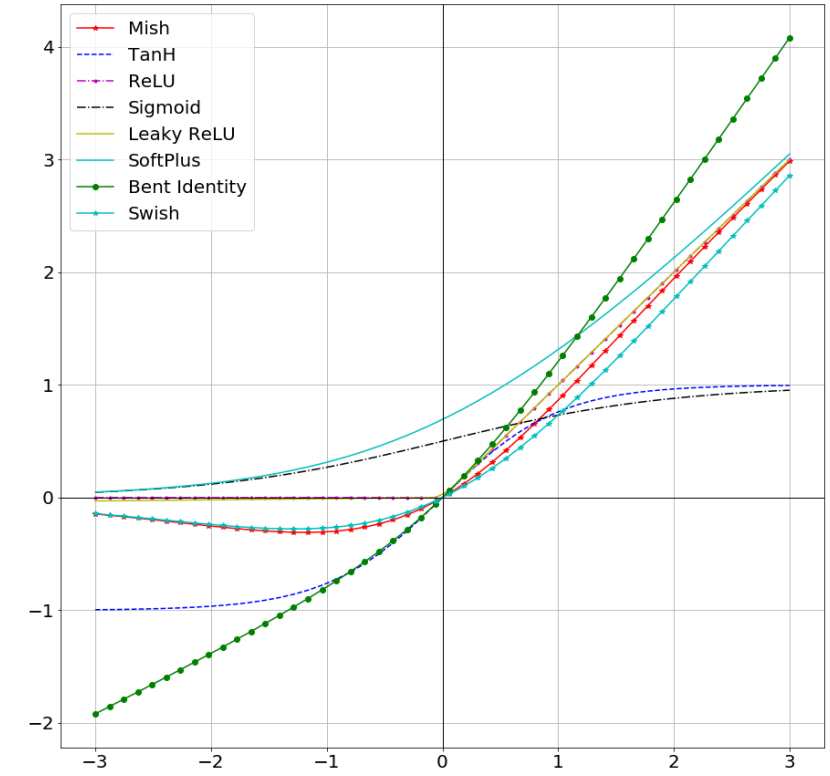
3. Activation Function

Non-linearity

- ReLU 제일 먼저 고려 -> 간단하면서 성능 굿
- Leaky ReLU, Parametric ReLU, ELU, GeLU 등 다양한 variation







4. Weight Initialization

- Neural Network 학습 전 가중치들을 초기화하는 과정
- Gradient Vanishing / Exploding 방지

- 일반적으로 작은 random value로 parameter의 초기값을 줘야 함
  - Hidden layer의 bias term은 0 으로 초기값 부여
  - 다른 weight 는 Uniform(-r, r)에서 sampling (r은 매우 작거나 크면 안됨)

- Xavier Initialization : n_{in}, n_{out} 에 맞게 weight variance 조정

$$\mathrm{Var}(W_i) = \frac{2}{n_{\mathrm{in}} + n_{\mathrm{out}}}$$

5. Optimizer

- **목적함수 f의 최솟값을 찾는 알고리즘**
- **보통 SGD 사용해도 최적화 good -> but 더 좋은 값을 얻기 위해서는 learning rate 튜닝**

- **복잡한 neural net 구조에서는 "adaptive" optimizer가 성능이 좋음 !**
  - gradient 정보를 계속 축적하여 이를 통해 gradient 조절하는 방법

- **Adam (Adaptive Optimizer) : 상대적인 업데이트 양에 따라 step size를 조정**
- **RAdam (Rectified Adam) : adaptive learning rate term의 분산을 rectify 함으로써, 학습의 안정성 확보**
- **Adagrad, RMSProp, …**

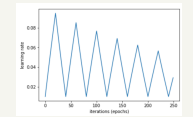
6. Learning Rate

- **0.01 정도의 일정한 learning rate 를 일반적으로 사용**
  - 보통 10의 배수로 ~!
  - 너무 작으면 학습이 느리고 ⇔ 너무 크면 발산

- **학습이 진행될수록 learning rate 를 감소시키는 방법이 good good**
  - k epoch 마다 반으로 줄이기
  - $lr = lr_0 e^{-kt}$
  - Cyclic learning rate : cyclic으로 주기를 만드는 방법

```python
# Learning Rate Scheduling (Performance Scheduling)
ReduceLROnPlateau(monitor='loss', mode="auto",
                  factor=0.25, min_delta=0.0001, cooldown=0, min_lr=0.00001)


# Cyclic Learning Rate
cyclic_learning_rate( global_step,  # Epoch
                      learning_rate=0.01,
                      max_lr=0.1,   # 최대 learning rate
                      step_size=20.,
                      gamma=0.99994,     # 작게 할수록 진폭이 더 크게 변한다
                      mode='triangular',
                      name=None):
```



Python

Reference

CS224n Winter 2019, Lecture04 : Backpropagation and Computation Graph

고려대학교 산업경영공학과 DSBA 연구실 CS224n Winter 2019 세미나 자료

투빅스 13기 이유민님 텍스트 세미나 강의자료

투빅스 13기 이지용님, 투빅스 12기 배유나님 Neural Network 기초 강의자료

투빅스 13기 김현선님 Neural Network 심화 강의자료

투빅스 13기 이예지님 RNN 강의자료

Transfer Learning (Pre-training vs Fine-Tuning) : https://analysisbugs.tistory.com/103 , https://jeinalog.tistory.com/13

Fine-Tuning : http://freesearch.pe.kr/archives/4963 , https://jeinalog.tistory.com/13

역전파 알고리즘의 효율 : http://norman3.github.io/prml/docs/chapter05/3.html (여기 블로그 좋으니깐 읽어보세용..)

들어주셔서 감사합니다.