

Lecture 04. Backpropagation and Computation Graphs

Css224n Natural language Processing with Deep Learning

UOS STAT NLP Study
Hyehyeon Moon

2020.11.04

[Lecture4_Backpropagation 요약]

1. 행렬로 표현한 Backpropagation(이전수업리뷰)
2. 개별변수로 표현한 Backpropagation
3. Word2Vec에서 Backpropagation을 이용해 weight을 업데이트시키는 방법
4. Wordvector를 Training 시켰을 때 일어날 수 있는 성능저하와 Pre-trained word vector와 Training word vector의 선택 상황제시
5. Backpropagation 계산(Single node, Multiple node, 코드)
6. Stuff you should know

-Regularization, Vectorization, Nonlinearities, Initialization, Optimizers,
Learning rates

- 2번 참고자료 : Multi Layer Perceptron에서 Backpropagation 공식(서울시립대 "인공지능" 강의_유하진 교수님)
- 5번 참고자료 : Backpropagation 연산자에 따른 gradient 계산방법 요약(서울시립대 "인공지능" 강의_유하진 교수님)

Natural Language Processing with Deep Learning

CS224N/Ling284



Christopher Manning
Lecture 4: Backpropagation and
computation graphs

Lecture Plan

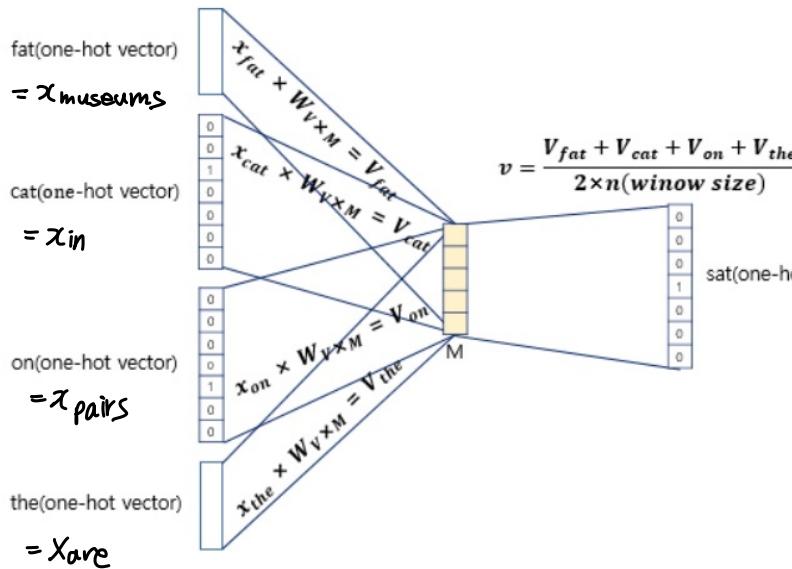
Lecture 4: Backpropagation and computation graphs → 학습표현

1. Matrix gradients for our simple neural net and some tips [15 mins]
2. Computation graphs and backpropagation [40 mins] (생략)
3. Stuff you should know [15 mins]
 - a. Regularization to prevent overfitting → overfitting: L_1, L_2 regularization
 - b. Vectorization → 속도개선
 - c. Nonlinearities → ReLU
 - d. Initialization → $V(w) = \frac{2}{in+out}$
 - e. Optimizers
 - f. Learning rates

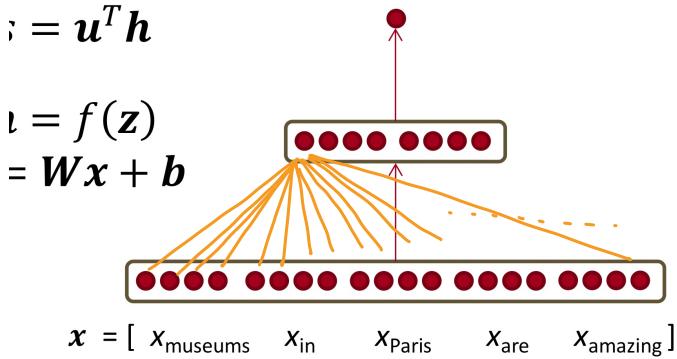
→ 상세하게 설명: 1번, 3-e, 3-f

딥러닝을 빼다른 표현으로 풀어

(i) Word2Vec 풀이



(ii) Softmax의 풀이



학습률 표현식은 Backpropagation

①

$$S = u^T h$$

$$h = f(z)$$

$$z = Wx + b$$

$$x = [x_{\text{museums}} \ x_{\text{in}} \ x_{\text{paris}} \ x_{\text{are}} \ x_{\text{amazing}}]$$

$$\frac{\partial S}{\partial b} = \frac{\partial S}{\partial h} \cdot \frac{\partial h}{\partial z} \cdot \frac{\partial z}{\partial b}$$

$$= u^T \cdot \text{diag}(f'(z)) I$$

$$= u^T \circ f'(z)$$

$$\left(\frac{\partial h}{\partial z} \right)_{ij} = \frac{\partial h_i}{\partial z_j} = \frac{\partial}{\partial z_j} f(z_i)$$

$$= \begin{cases} f'(z_i) & i=j \\ 0 & i \neq j \end{cases}$$

$$\Rightarrow \frac{\partial h}{\partial z} = \begin{pmatrix} f'(z_1) & & & & 0 \\ & \ddots & & & \\ 0 & & \ddots & & f'(z_n) \end{pmatrix}$$

$$= \text{diag}(f'(z))$$

② δ : local error signal

$$\frac{\partial S}{\partial w} = \frac{\partial S}{\partial h} \frac{\partial h}{\partial z} \frac{\partial z}{\partial w} = \delta \frac{\partial z}{\partial w}$$

$$\frac{\partial S}{\partial b} = \frac{\partial S}{\partial h} \frac{\partial h}{\partial z} \frac{\partial z}{\partial b} = \delta \frac{\partial z}{\partial b}$$

$$\Rightarrow \delta = \frac{\partial S}{\partial h} \frac{\partial h}{\partial z} = u^T \circ f'(z)$$

1x6

* Jacobian matrix (자료)

$f(x) = [f_1(x_1, x_2, \dots, x_n), \dots, f_m(x_1, x_2, \dots, x_n)]$: m output and n input

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \quad \left(\frac{\partial f}{\partial x} \right)_{ij} = \frac{\partial f_i}{\partial x_j}$$

③ $\frac{\partial S}{\partial w}$ 의 이론 결과는?

$$\theta^{\text{new}} = \theta^{\text{old}} - \alpha \nabla_{\theta} J(\theta)$$
 를 풀어쓰면 됨

$$\nabla_{\theta} J(\theta) = \frac{\partial S}{\partial \theta}$$
 6x20 험태 1x10 학습률

$$\Rightarrow \frac{\partial S}{\partial w} = \delta x$$
 인데 6x20 험태를 만들어주기 위해 $\frac{\partial S}{\partial w} = \delta^T x^T$

$\begin{bmatrix} \partial S \\ \partial w \end{bmatrix}$ $\begin{bmatrix} \delta \\ x \end{bmatrix}$ $\begin{bmatrix} 1 \\ x^T \end{bmatrix}$
6x6 6x1 1x20

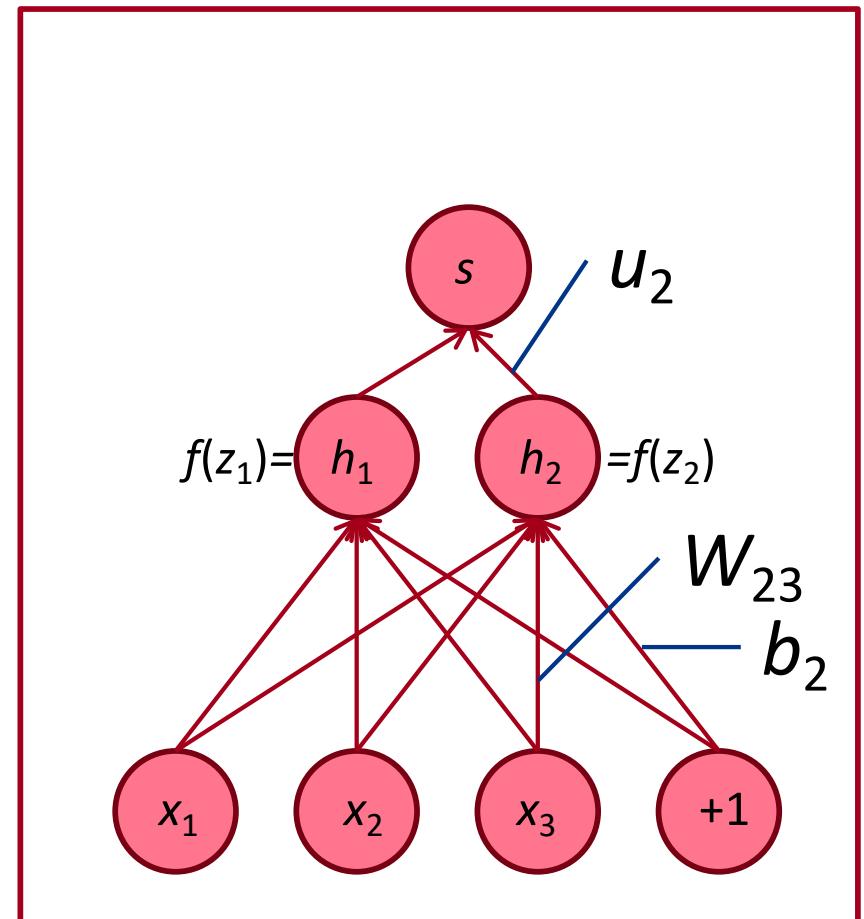
Deriving gradients for backprop

- For this function (following on from last time):

$$\frac{\partial s}{\partial W} = \delta \frac{\partial z}{\partial W} = \delta \frac{\partial}{\partial W}(Wx + b)$$

- Let's consider the derivative of a single weight W_{ij}
- W_{ij} only contributes to z_i
 - For example: W_{23} is only used to compute z_2 not z_1

$$\begin{aligned}\frac{\partial z_i}{\partial W_{ij}} &= \frac{\partial}{\partial W_{ij}}(W_{i \cdot} \cdot x + b_i) \\ &= \frac{\partial}{\partial W_{ij}} \sum_{k=1}^d W_{ik} x_k = x_j\end{aligned}$$



Deriving gradients for backprop

- So for derivative of single W_{ij} :

$$\frac{\partial s}{\partial W_{ij}} = \delta_i x_j$$

→ 벡터표현이 아닌 단일로 봤을 때
아래 슬라이드

벡터표현 $\frac{\partial s}{\partial w_{ij}} = \underbrace{\frac{\partial s}{\partial h_i} \frac{\partial h_i}{\partial z_i} \frac{\partial z_i}{\partial w_{ij}}}_{= \delta_i x_j}$

Error signal from above Local gradient signal

- We want gradient for full W – but each case is the same
- Overall answer: Outer product:

$$\frac{\partial s}{\partial W} = \boldsymbol{\delta}^T \quad \mathbf{x}^T$$
$$[n \times m] \quad [n \times 1][1 \times m]$$

define

$$y_i = f(\text{net}_i)$$

$$E = \frac{1}{2}(t-y)^2$$

$$\frac{\partial E}{\partial y} \frac{\partial y}{\partial \text{net}}$$

$$\delta_j = -\frac{\partial E}{\partial \text{net}_j}$$

$$\frac{\partial y_i}{\partial \text{net}_i} = f'(\text{net}_i)$$

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial y_i} \frac{\partial y_i}{\partial \text{net}_i} \frac{\partial \text{net}_i}{\partial w_{ki}}$$

$$\frac{\partial \text{net}_j}{\partial y_i} = w_{ij}$$

$$\delta_i = -\frac{\partial E}{\partial \text{net}_i} \frac{\partial \text{net}_i}{\partial w_{ki}} = x_k$$

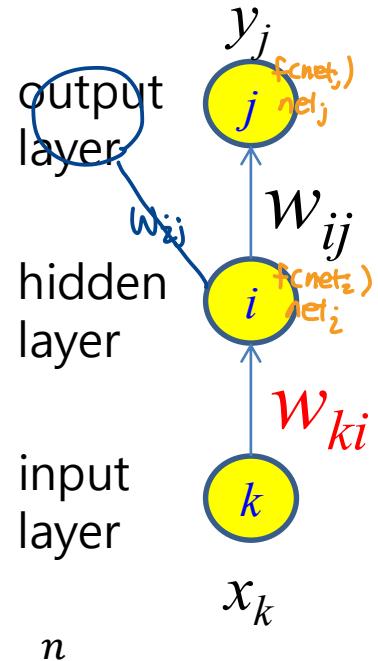
$$\text{net}_j = \sum_{i=0}^n y_i w_{ij}$$

$$\text{net}_i = \sum_{i=0}^n x_k w_{ki}$$

$$\frac{\partial E}{\partial w_{ki}} = -\delta_j w_{ij} f'(\text{net}_i) x_k = -\delta_i x_k$$

$$\Delta w_{ki} = c f'(\text{net}_i) x_k \sum_j \delta_j w_{ij} = c \delta_i x_k$$

$$\delta_i = f'(\text{net}_i) \sum_j \delta_j w_{ij}$$



[Word2Vec에서 Backpropagation을 이용해 weight를 업데이트시키는 방법]

Deriving gradients wrt words for window model

- The gradient that arrives at and updates the word vectors can simply be split up for each word vector:
- Let $\nabla_x J = W^T \delta = \delta_{x_{window}}$  ???
- With $x_{window} = [x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing}]$
- We have

$$\delta_{window} = \begin{bmatrix} \nabla x_{museums} \\ \nabla x_{in} \\ \nabla x_{Paris} \\ \nabla x_{are} \\ \nabla x_{amazing} \end{bmatrix} \in \mathbb{R}^{5d}$$

문제: Is we can just sort-of split this window vector into five pieces and say

We have five updates to word vectors

$\underbrace{W_{v \times M}^T}_\text{5개이므로} \cdot \underbrace{x_{\text{window}}}_\text{5개이므로}$, input 단어가 5개이므로 x_{window}

window에 같은 단어가 있으면 동일하게 업데이트

하나요.

Ex) Cow-bow

fat(one-hot vector)

$= x_{\text{museums}}$

cat(one-hot vector)

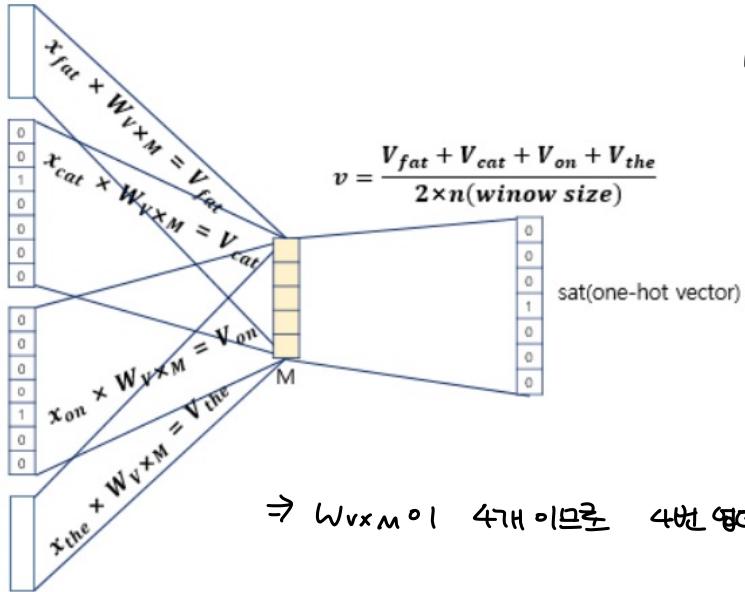
$= x_{in}$

on(one-hot vector)

$= x_{paris}$

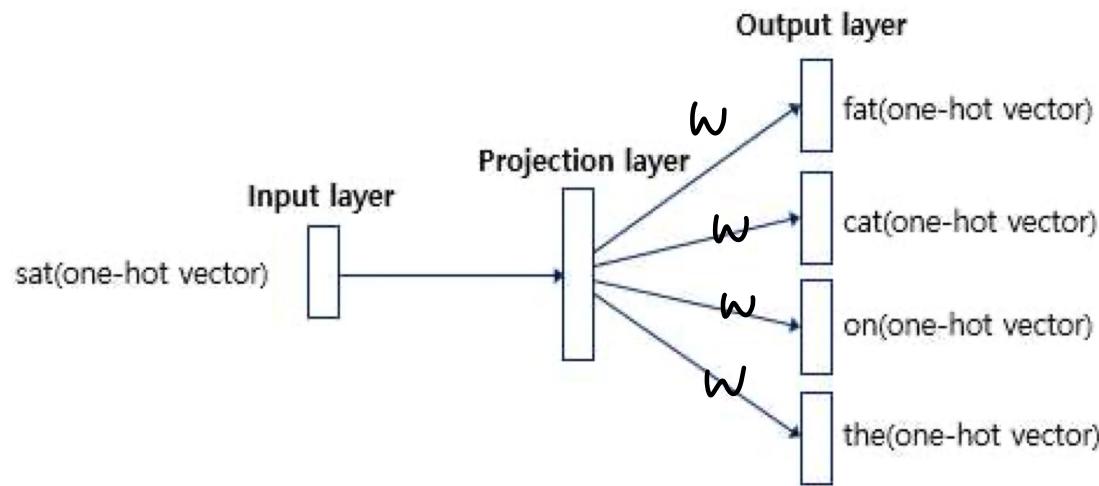
the(one-hot vector)

$= x_{are}$



$\Rightarrow W_{v \times M}^T$ 4개이므로 4개의 업데이트를 해주어야 = $\frac{1}{4}$

ex) skip-gram



어떤 단어 → 단어에 부여된 고유한 정수값 → 임베딩 층 통과 → 밀집 벡터

임베딩 층은 입력 정수에 대해 밀집 벡터(dense vector)로 맵핑하고 이 밀집 벡터는 인공 신경망의 학습 과정에서 가중치가 학습되는 것과 같은 방식으로 훈련됩니다. 훈련 과정에서 단어는 모델이 풀고자하는 작업에 맞는 값으로 업데이트 됩니다. 그리고 이 밀집 벡터를 임베딩 벡터라고 부릅니다.

정수를 밀집 벡터 또는 임베딩 벡터로 맵핑한다는 것은 어떤 의미일까요? 특정 단어와 맵핑되는 정수를 인덱스로 가지는 테이블로부터 임베딩 벡터 값을 가져오는 룩업 테이블이라고 볼 수 있습니다. 그리고 이 테이블은 단어 집합의 크기만큼의 행을 가지므로 모든 단어는 고유한 임베딩 벡터를 가집니다.

Word → Integer → lookup Table → Embedding vector



위의 그림은 단어 great이 정수 인코딩 된 후 테이블로부터 해당 인덱스에 위치한 임베딩 벡터를 꺼내오는 모습을 보여줍니다. 위의 그림에서는 임베딩 벡터의 차원이 4로 설정되어져 있습니다. 그리고 단어 great은 정수 인코딩 과정에서 1,918의 정수로 인코딩이 되었고 그에 따라 단어 집합의 크기만큼의 행을 가지는 테이블에서 인덱스 1,918번에 위치한 행을 단어 great의 임베딩 벡터로 사용합니다. 이 임베딩 벡터는 모델의 입력이 되고, 역전파 과정에서 단어 great의 임베딩 벡터값이 학습됩니다.

룩업 테이블의 개념을 이론적으로 우선 접하고, 처음 케라스를 배울 때 어떤 분들은 임베딩 층의 입력이 원-핫 벡터가 아니어도 동작한다는 점에 헷갈려 합니다. 케라스는 단어를 정수 인덱스로 바꾸고 원-핫 벡터로 한번 더 바꾸고나서 임베딩 층의 입력으로 사용하는 것이 아니라, 단어를 정수 인덱스로만 바꾼채로 임베딩 층의 입력으로 사용해도 룩업 테이블 된 결과인 임베딩 벡터를 리턴합니다.

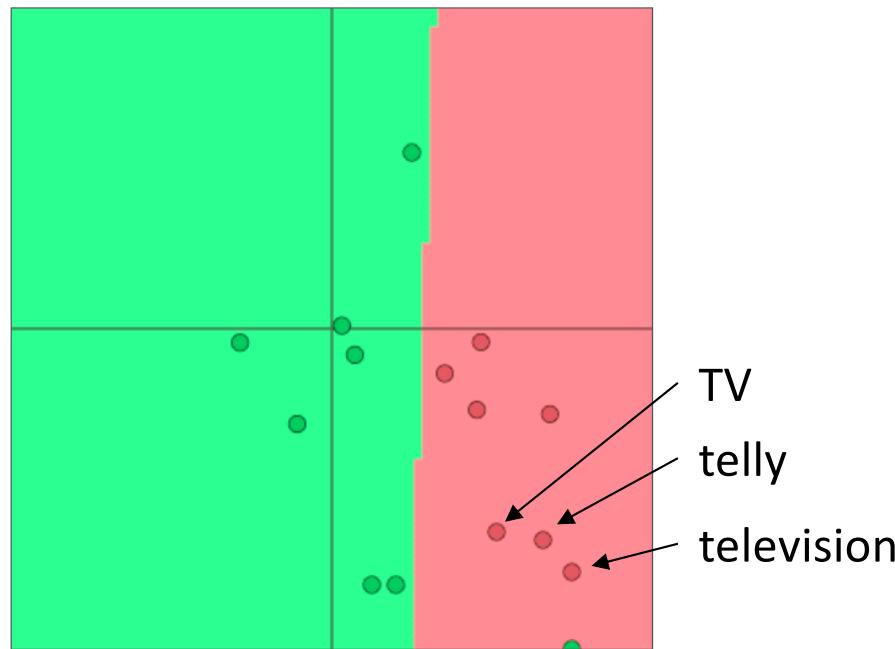
Updating word gradients in window model

- This will push word vectors around so that they will (in principle) be more helpful in determining named entities.
- For example, the model can learn that seeing x_{in} as the word just before the center word is indicative for the center word to be a location

[Word vector 를 training 시켰을 때 일어날 수 있는 성능 저하]

A pitfall when retraining word vectors

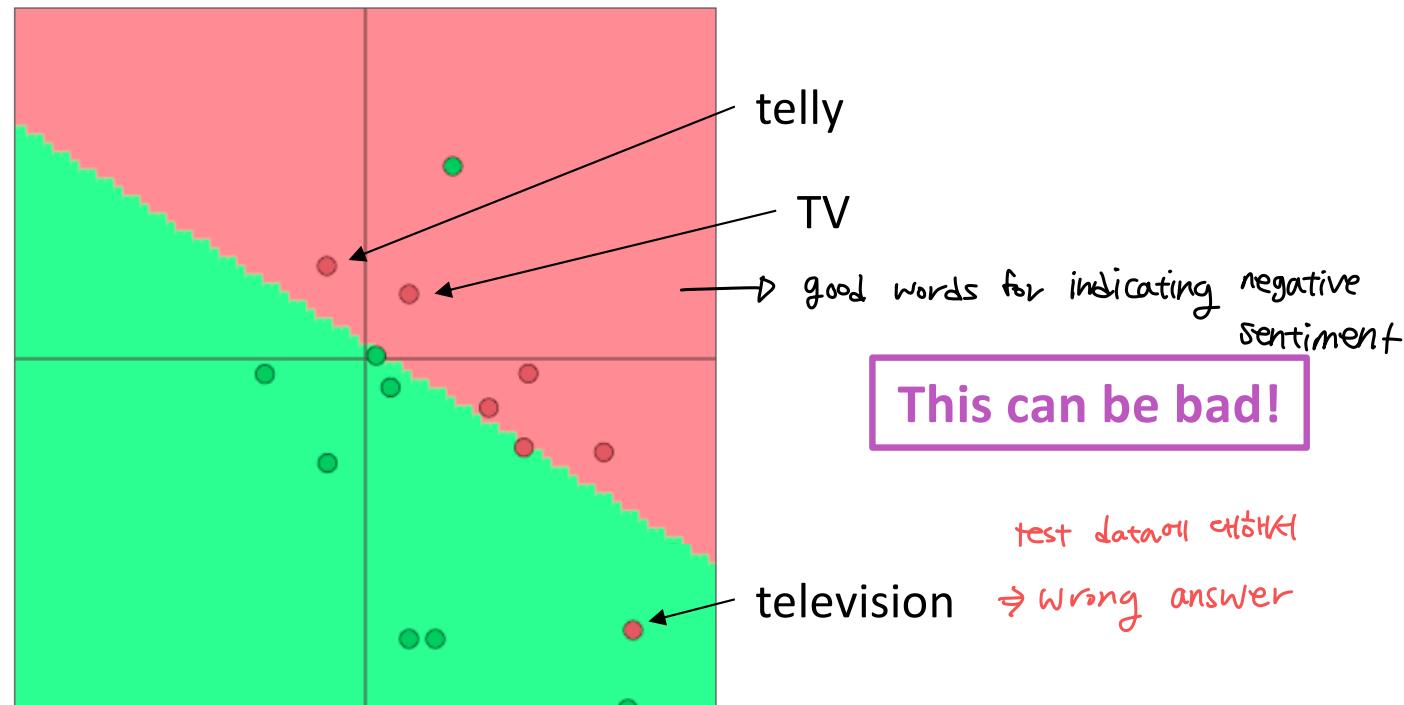
- **Setting:** We are training a logistic regression classification model for movie review sentiment using single words.
- In the **training data** we have “TV” and “telly”
- In the **testing data** we have “television”
- The **pre-trained** word vectors have all three similar:



- **Question:** What happens when we update the word vectors?

A pitfall when retraining word vectors

- **Question:** What happens when we update the word vectors?
- **Answer:**
 - Those words that are **in** the training data **move around**
 - “TV” and “telly”
 - Words **not** in the training data **stay where they were**
 - “television”



So what should I do?

- **Question:** Should I use available “pre-trained” word vectors

Answer:

- Almost always, yes!
- They are trained on a huge amount of data, and so they will know about words not in your training data and will know more about words that are in your training data
- Have 100s of millions of words of data? Okay to start random

- **Question:** Should I update (“fine tune”) my own word vectors?

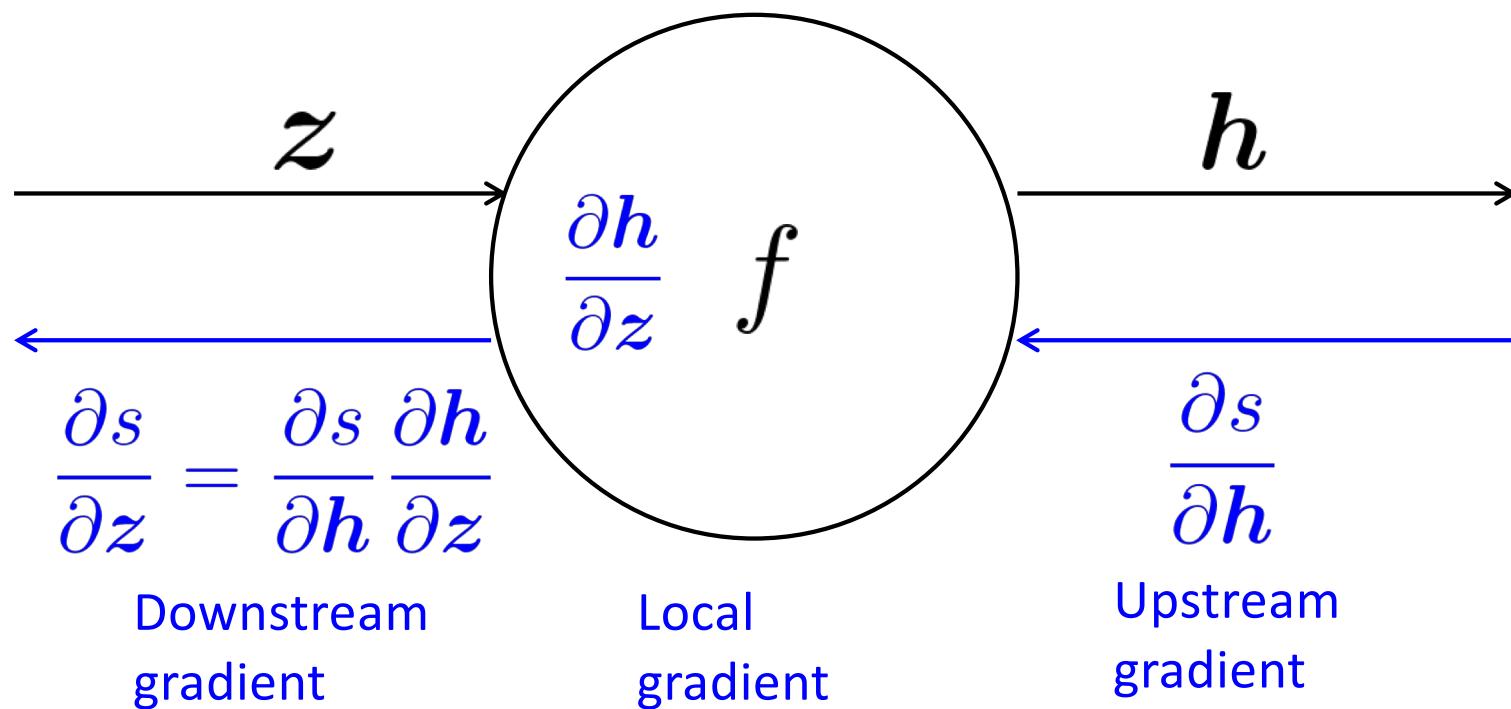
Answer:

- If you only have a **small** training data set, **don't** train the word vectors
- If you have have a **large** dataset, it probably will work better to **train = update = fine-tune** word vectors to the task

Backpropagation: Single Node

- Each node has a **local gradient**
 - The gradient of it's output with respect to it's input
- [downstream gradient] = [upstream gradient] x [local gradient]

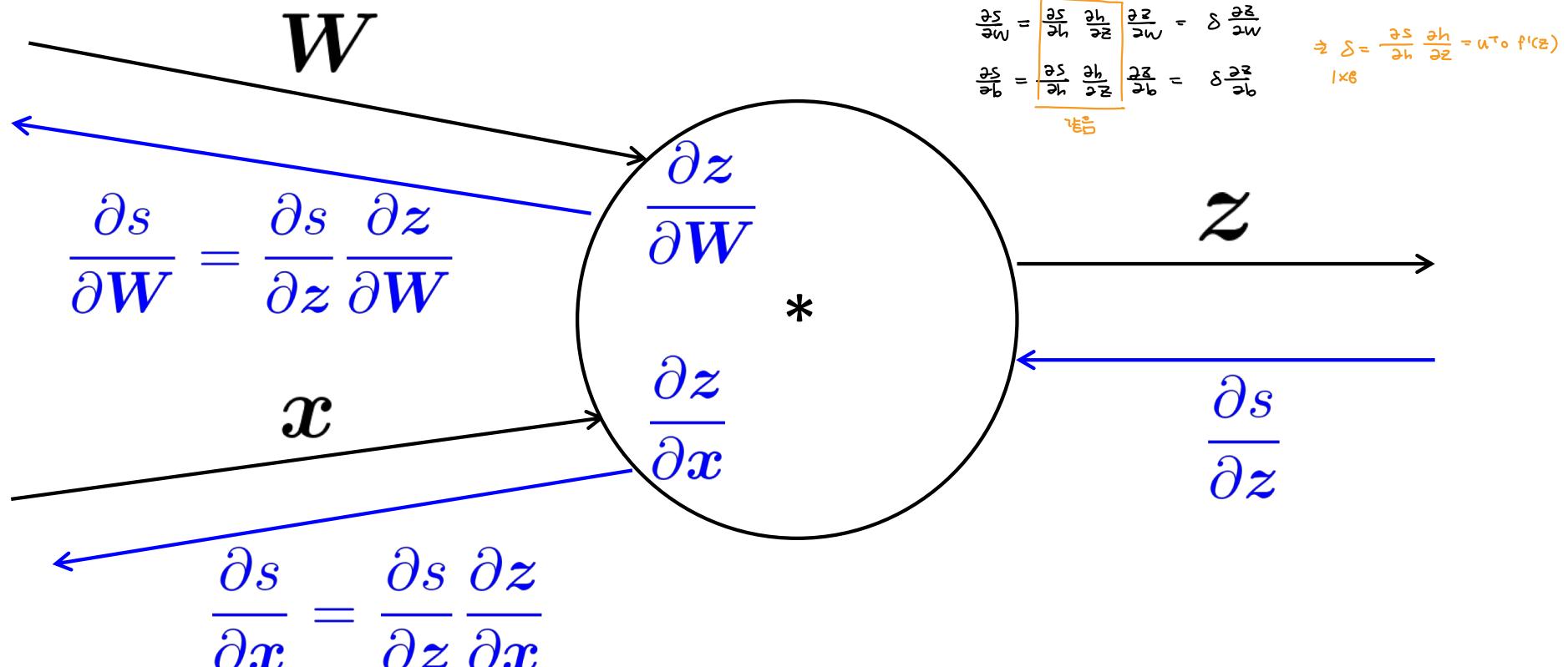
$$h = f(z)$$



Backpropagation: Single Node

- Multiple inputs → multiple local gradients

$$z = Wx$$



An Example

$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$

$$b = \max(y, z)$$

$$f = ab$$

$$\frac{\partial f}{\partial x} = 2$$

$$\frac{\partial f}{\partial y} = 3 + 2 = 5$$

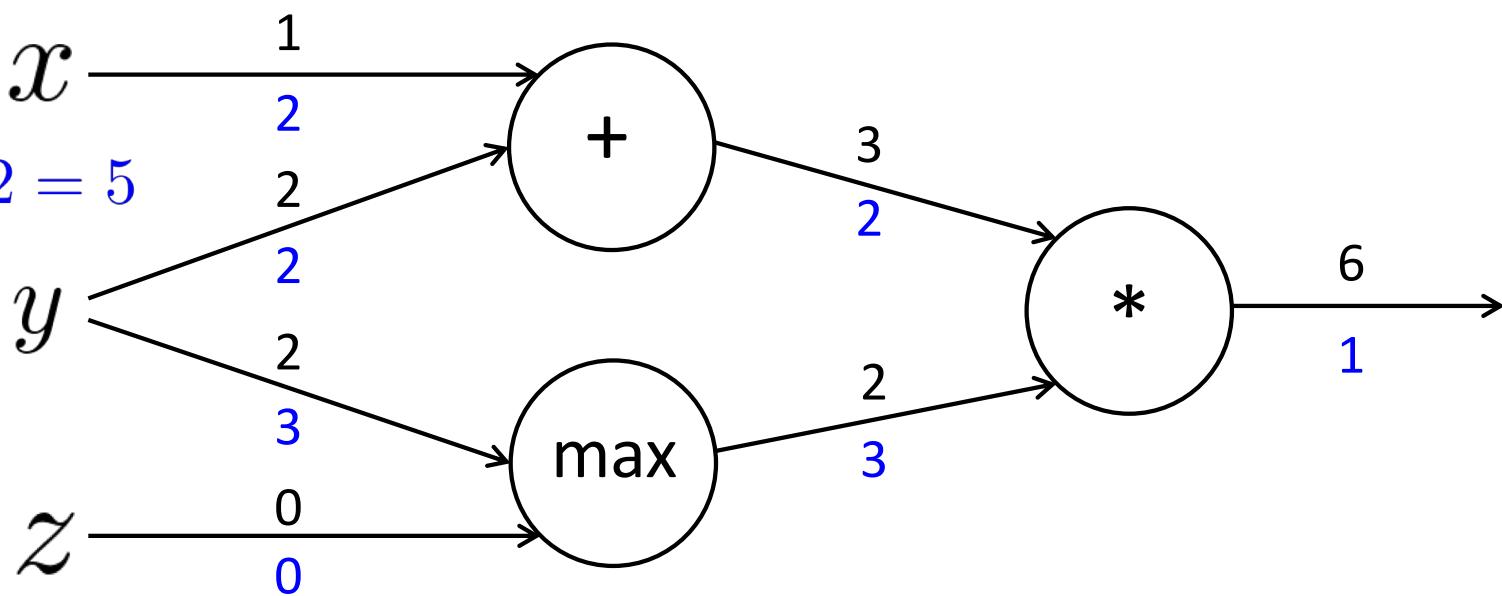
$$\frac{\partial f}{\partial z} = 0$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$

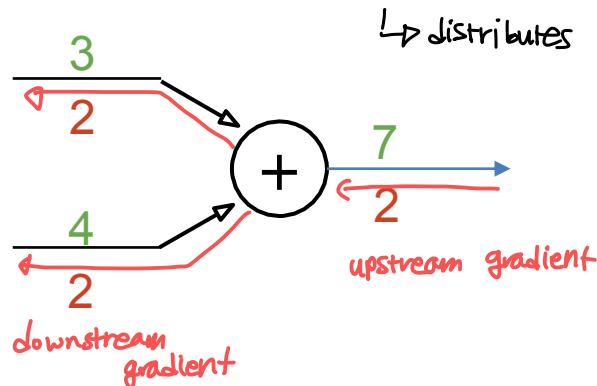
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$

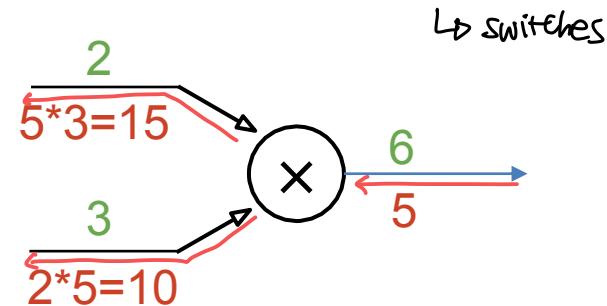


Patterns in gradient flow

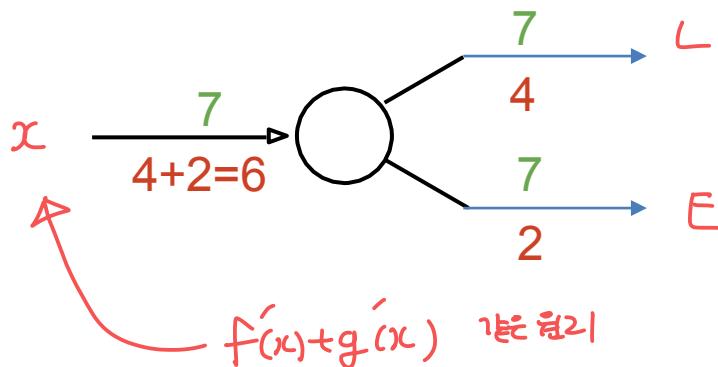
add gate: gradient distributor



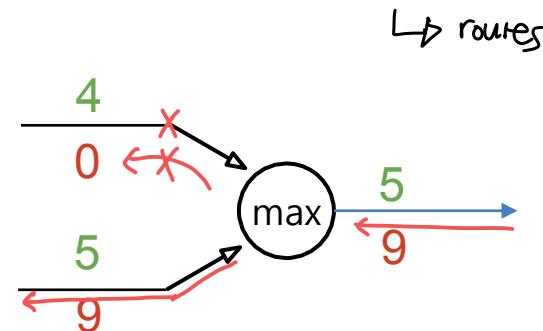
mul gate: "swap multiplier"



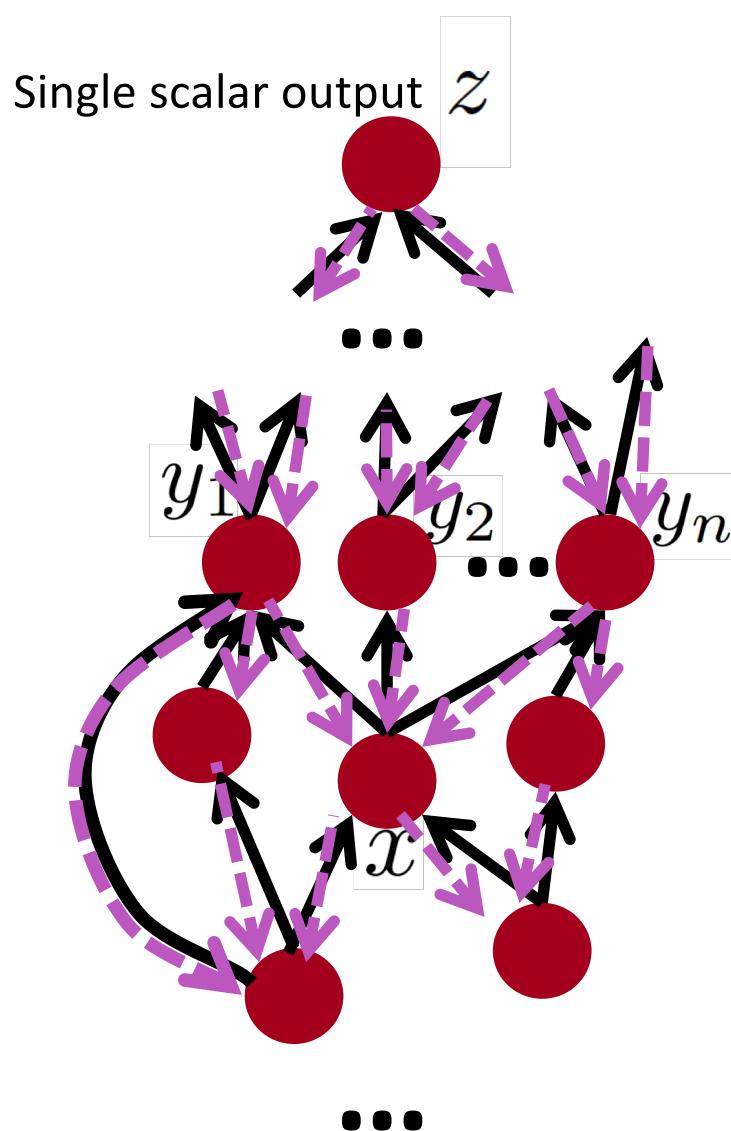
copy gate: gradient adder



max gate: gradient router



Back-Prop in General Computation Graph



우상학적인

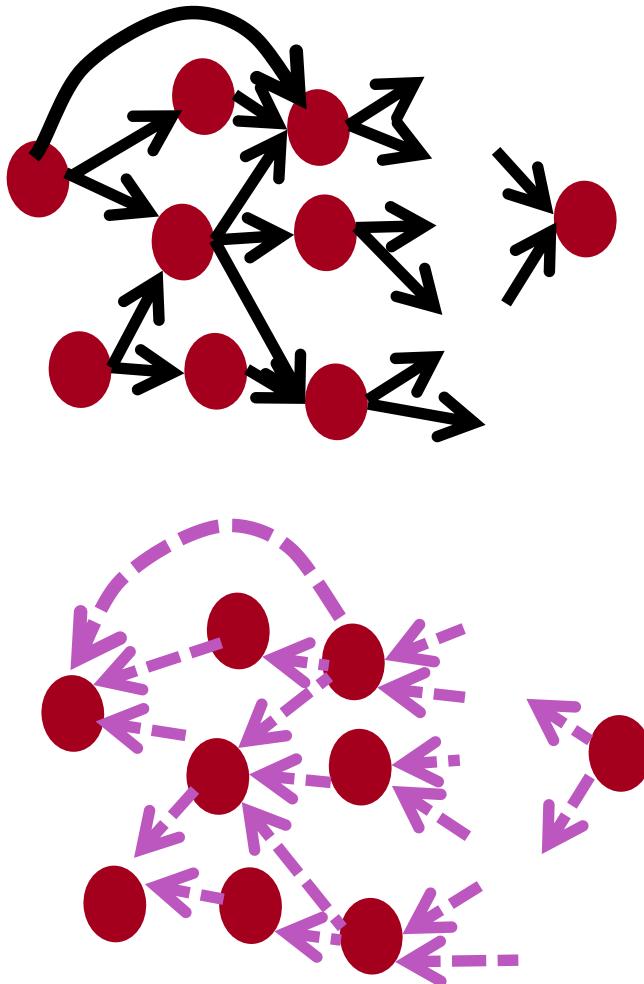
1. Fprop: visit nodes in topological sort order
 - Compute value of node given predecessors
 2. Bprop:
 - initialize output gradient = 1
 - visit nodes in reverse order:
Compute gradient wrt each node using gradient wrt successors
- $\{y_1, y_2, \dots, y_n\}$ = successors of x

$$\frac{\partial z}{\partial x} = \sum_{i=1}^n \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Done correctly, big $O()$ complexity of fprop and bprop is **the same**

In general our nets have regular layer-structure and so we can use matrices and Jacobians...

Automatic Differentiation



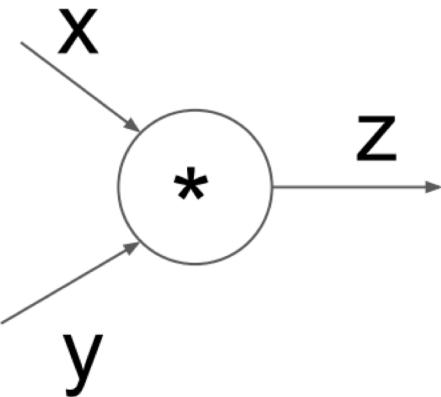
- The gradient computation can be automatically inferred from the symbolic expression of the fprop
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output
- Modern DL frameworks (Tensorflow, PyTorch, etc.) do backpropagation for you but mainly leave layer/node writer to hand-calculate the local derivative

↳ modern DL framework는 개별 노드에 대한
이론값을 주지 않음.

Backprop Implementations

```
class ComputationalGraph(object):  
    #...  
    def forward(inputs):  
        # 1. [pass inputs to input gates...]  
        # 2. forward the computational graph:  
        for gate in self.graph.nodes_topologically_sorted():  
            gate.forward()  
        return loss # the final gate in the graph outputs the loss  
    def backward():  
        for gate in reversed(self.graph.nodes_topologically_sorted()):  
            gate.backward() # little piece of backprop (chain rule applied)  
        return inputs_gradients
```

Implementation: forward/backward API



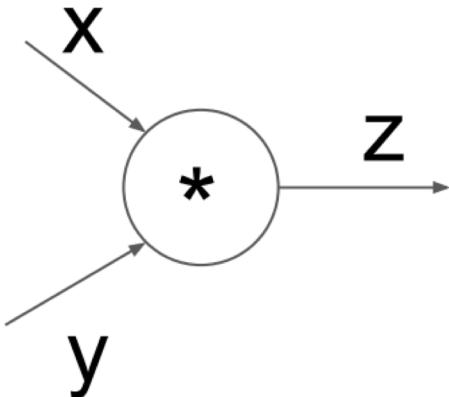
(**x,y,z** are scalars)

```
class MultiplyGate(object):  
    def forward(x,y):  
        z = x*y  
        return z  
    def backward(dz):  
        # dx = ... #todo  
        # dy = ... #todo  
        return [dx, dy]
```

$$\frac{\partial L}{\partial z}$$

$$\frac{\partial L}{\partial x}$$

Implementation: forward/backward API



(x, y, z are scalars)

```
class MultiplyGate(object):  
    def forward(x,y):  
        z = x*y  
        self.x = x # must keep these around!  
        self.y = y  
        return z  
  
    def backward(dz):  
        local gradient dx = upstream gradient self.y * dz # [dz/dx * dL/dz]  
        dy = self.x * dz # [dz/dy * dL/dz]  
        return [dx, dy]
```

↳ 이미 많은 사람들이 코드를 알맞게 만들었기 때문에

연구 아닌 이상 딱 필요 없고, 너을 그만 들어가지 않아도 된다.

Gradient checking: Numeric Gradient

- For small h ($\approx 1e-4$), $f'(x) \approx \frac{f(x + h) - f(x - h)}{2h}$
- Easy to implement correctly
- But approximate and **very slow**:
 - Have to recompute f for **every parameter** of our model
- **Useful for checking your implementation**
 - In the old days when we hand-wrote everything, it was key to do this everywhere.
 - Now much less needed, when throwing together layers

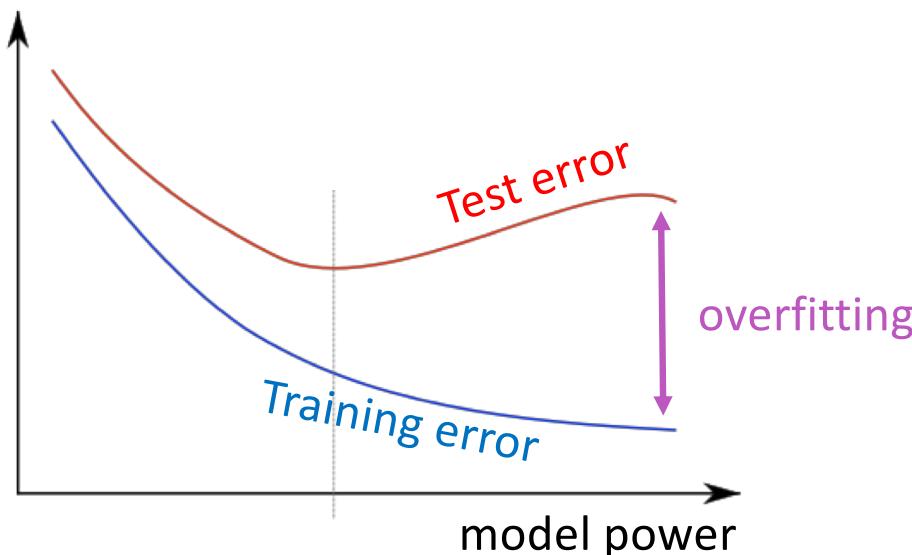
3. We have models with many params! Regularization!

- Really a full loss function in practice includes **regularization** over all parameters θ , e.g., L2 regularization:

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

正則化
You're going to be penalized to the extent that you move parameters away from zero

- Regularization (largely) prevents **overfitting** when we have a lot of features (or later a very powerful/deep model, ++)



“Vectorization”

- E.g., looping over word vectors versus concatenating them all into one large matrix and then multiplying the softmax weights with that matrix

```
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)      5x300
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)  300x500

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

- 1000 loops, best of 3: **639 µs** per loop
10000 loops, best of 3: **53.8 µs** per loop

“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)

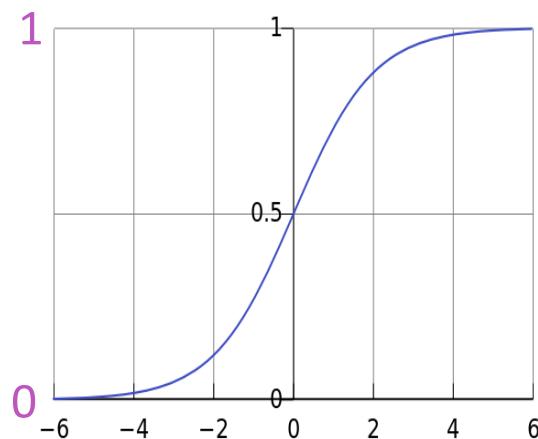
%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

- The (10x) faster method is using a C x N matrix
- Always try to use vectors and matrices rather than for loops!
- You should speed-test your code a lot too!!
- tl;dr: Matrices are awesome!!!

Non-linearities: The starting points

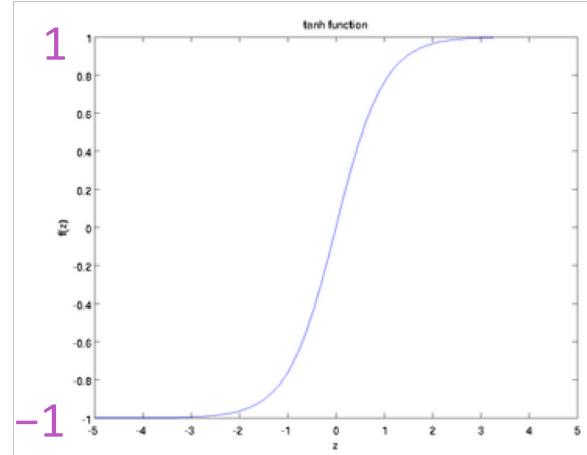
logistic (“sigmoid”)

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



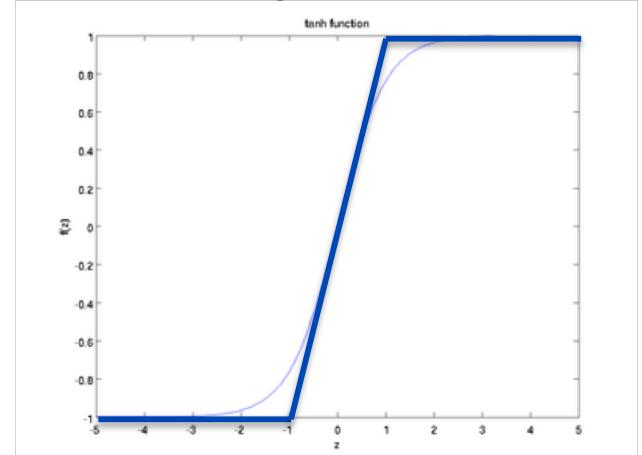
tanh

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



hard tanh

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



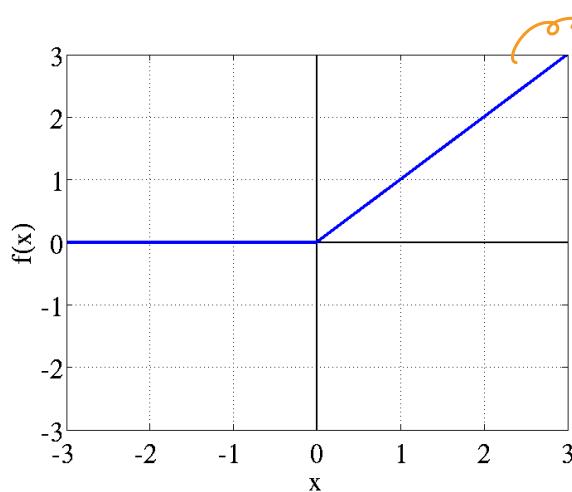
tanh is just a rescaled and shifted sigmoid ($2 \times$ as steep, $[-1,1]$):

$$\tanh(z) = 2\text{logistic}(2z) - 1$$

Both logistic and tanh are still used in particular uses, but are no longer the defaults for making deep networks

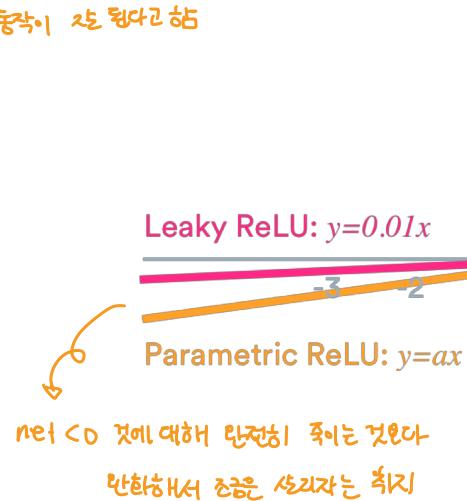
Non-linearities: The new world order

ReLU (rectified
linear unit) ~~hard tanh~~
 $\text{rect}(z) = \max(z, 0)$



Leaky ReLU

Parametric ReLU



- For building a feed-forward deep network, the first thing you should try is ReLU — it trains quickly and performs well due to good gradient backflow

Parameter Initialization

- You normally must initialize weights to small random values
 - To avoid symmetries that prevent learning/specialization
- Initialize hidden layer biases to 0 and output (or reconstruction) biases to optimal value if weights were 0 (e.g., mean target or inverse sigmoid of mean target)
- Initialize all other weights $\sim \text{Uniform}(-r, r)$, with r chosen so numbers get neither too big or too small
- Xavier initialization has variance inversely proportional to fan-in n_{in} (previous layer size) and fan-out n_{out} (next layer size):

많이 쓰임

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

→ 데이터가 정규분포를 가정할 경우에 허용해

정규분포 가정 허용에 많이 사용

Optimizers

- Usually, plain SGD will work just fine
 - However, getting good results will often require hand-tuning the learning rate (next slide)
- For more complex nets and situations, or just to avoid worry, you often do better with one of a family of more sophisticated “adaptive” optimizers that scale the parameter adjustment by an accumulated gradient.
 - These models give per-parameter learning rates
 - Adagrad
 - RMSprop
 - Adam ← A fairly good, safe place to begin in many cases
 - SparseAdam
 - ...

Learning Rates

- You can just use a constant learning rate. Start around $lr = 0.001$?
 - It must be order of magnitude right – try powers of 10
 - Too big: model may diverge or not converge
 - Too small: your model may not have trained by the deadline
- Better results can generally be obtained by allowing learning rates to decrease as you train
 - By hand: halve the learning rate every k epochs
 - An epoch = a pass through the data (shuffled or sampled)
 - By a formula: $lr = lr_0 e^{-kt}$, for epoch t
 - There are fancier methods like cyclic learning rates (q.v.)
- Fancier optimizers still use a learning rate but it may be an initial rate that the optimizer shrinks – so may be able to start high

ex) Adam

[Optimizers 추가공부]

1. Momentum
2. Nesterov Accelerated Gradient(NAG)
3. Adagrad
4. Adadelta
5. Rmsprop
6. Adam

- 자료출처 : 서울서울시립대 "인공지능" 강의_유하진 교수님
- 그 외 다수의 사이트 참고

Optimizers



서울시립대학교
UNIVERSITY OF SEOUL

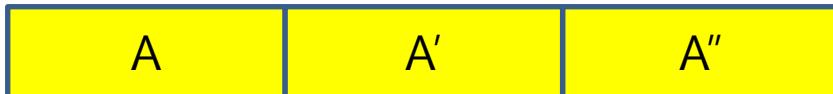
Optimizers – weight update 방법

- Batch and Minibatch Algorithms
- Stochastic gradient descent
- 학습률(ϵ)의 결정 방법
 - Learning rate decay
- Momentum ✓
- Nesterov Accelerated Gradient (NAG)
- Rmsprop
- Adagrad
- Adadelta
- Adam
- Adamax
- Nadam

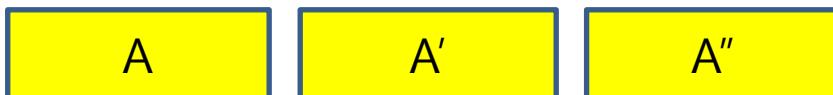
Stochastic gradient descent의 장점

\Leftrightarrow mini batch size의 gradient descent를 활용한 weight 업데이트 방법

- 훈련 데이터에 중복성이 있을 때 효율이 향상되고 학습이 빨리 진행된다

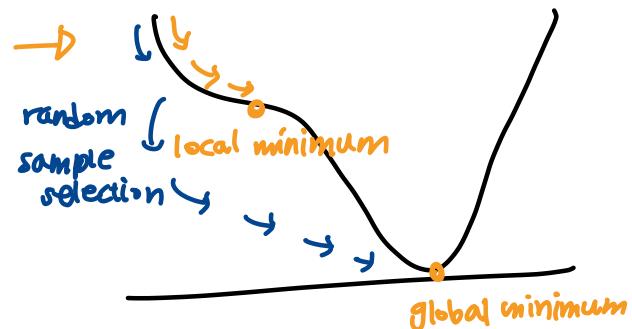


전체 데이터



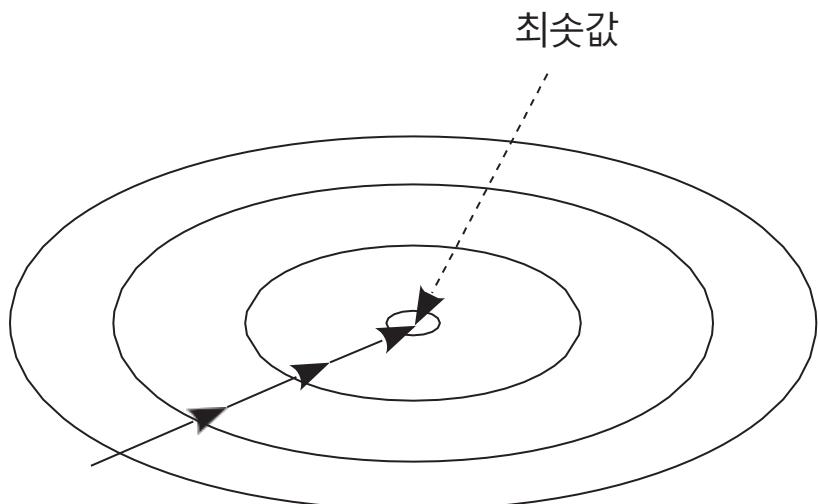
- Local minimum에 빠질 위험이 줄어든다
 - 반복할 때마다 random sample selection
- Update의 크기가 작은 상태로 학습이 진행
 - 학습의 경과 관찰이 용이

Stochastic gradient descent (sgd)

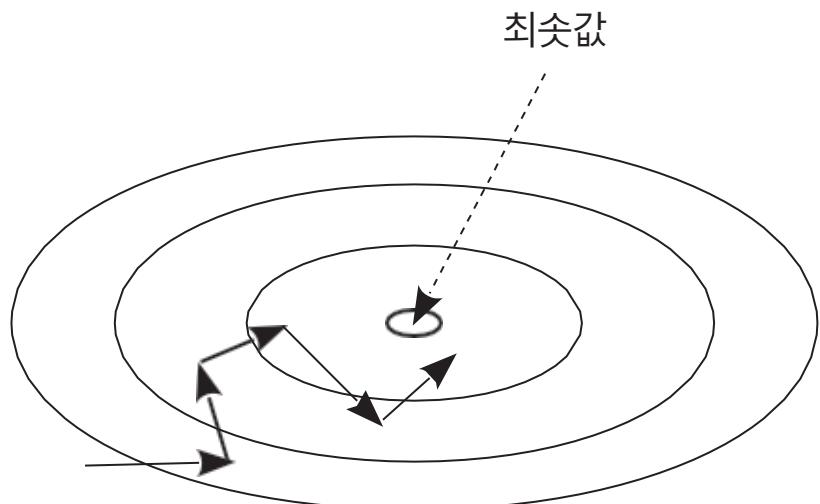


Stochastic gradient descent

- 기울기 벡터가 정확하게 계산되지 않음
- Avoid local minimum



모든 데이터를 사용한 경사 하강법



미니 배치에 의한 확률적 경사 하강법

Stochastic gradient descent 방법으로 최솟값으로 향하는 모습

출처: 딥 러닝 제대로 시작하기

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ε .

Require: Initial parameter θ

while stopping criterion not met do

 Sample a minibatch of m examples from the training set $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.

 Compute gradient estimate: $g \leftarrow 1/m \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

 Apply update: $\theta \leftarrow \theta - \varepsilon g$

end while

기울기

$$x^{(i)} = [x_0, x_1, \dots, x_D]$$

$f(x^{(i)}; \theta)$ = network output

$$\theta = [w_0, w_1, \dots, w_N]$$

$L(f(x^{(i)}; \theta), y^{(i)})$: Loss

$$\Delta\theta = [\Delta w_0, \Delta w_1, \dots, \Delta w_N]$$

$$g \leftarrow 1/m \nabla_{\theta} L() = [\partial L / \partial w_0, \dots, \partial L / \partial w_N]$$



가속도 : $\frac{\text{속도의 변화량}}{\text{단위시간}} = \frac{\text{weight 변화량}}{1 \text{ epoch}}$

Momentum – 학습속도 향상

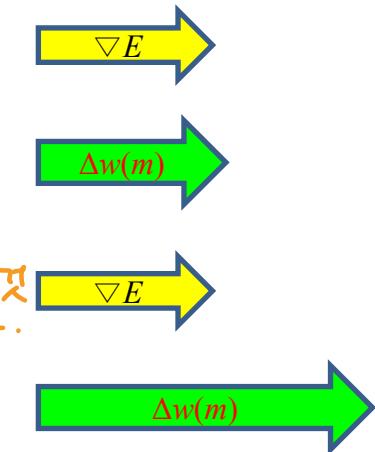
- 가속도 (gradient 가 속도를 조절하는 역할)
- 이전 weight 변화(속도)를 어느 정도 유지 → 학습속도 ↑

$$\Delta w^{(t)} = \alpha \Delta w^{(t-1)} - \epsilon \nabla E_t$$

Learning rate

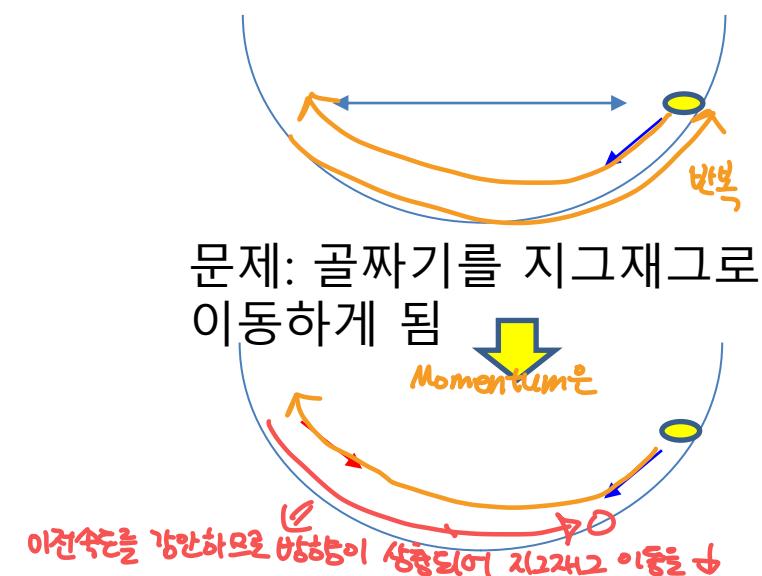
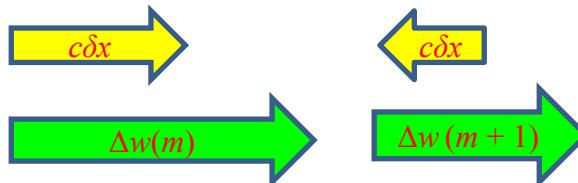
↑ 이전 변화량 더해짐
↓ 이전 기울기
Momentum constant

어느 방향으로 가는 것을
따라서 가지게 만드는 것
→ → → ..

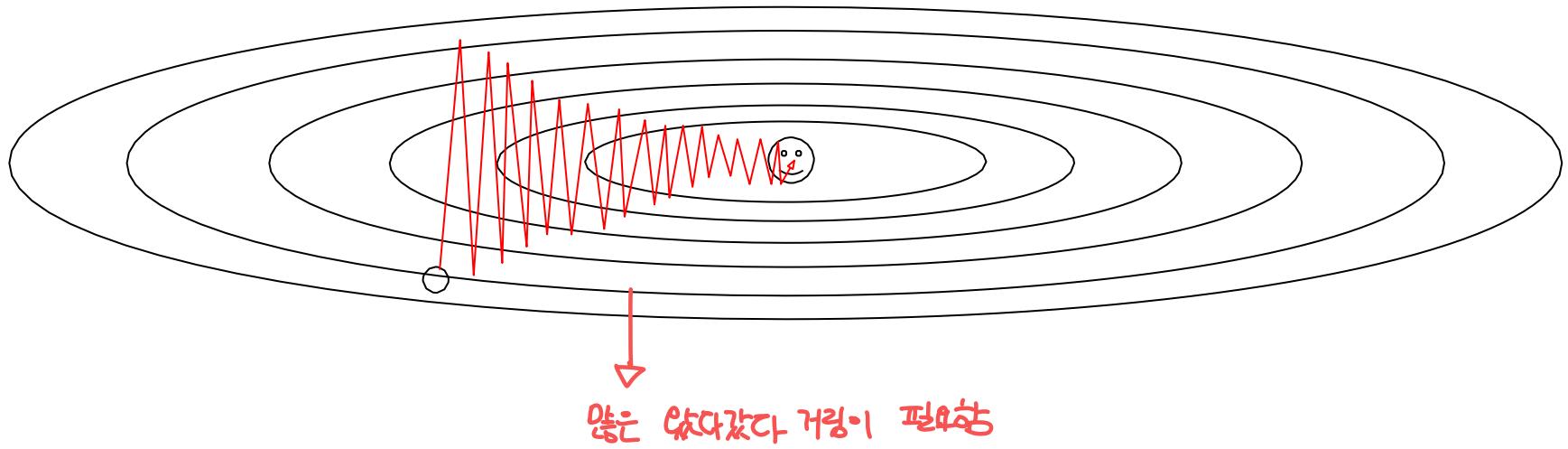


- . 이전 변화량 (속도 v)

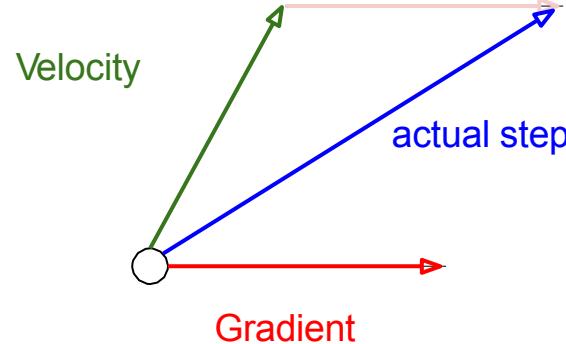
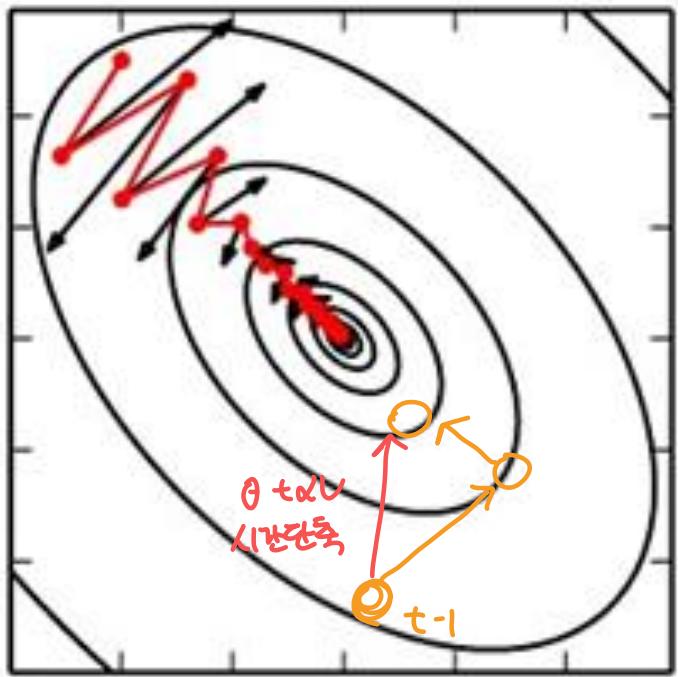
- weight의 업데이트 값에 이전 업데이트 값의 일정 비율을 더함.
- $\alpha = 0.5 \sim 0.9$
- 효과:
 - 학습속도 향상
 - Weight 변화가 oscillating 하는 것 방지



Problems with SGD

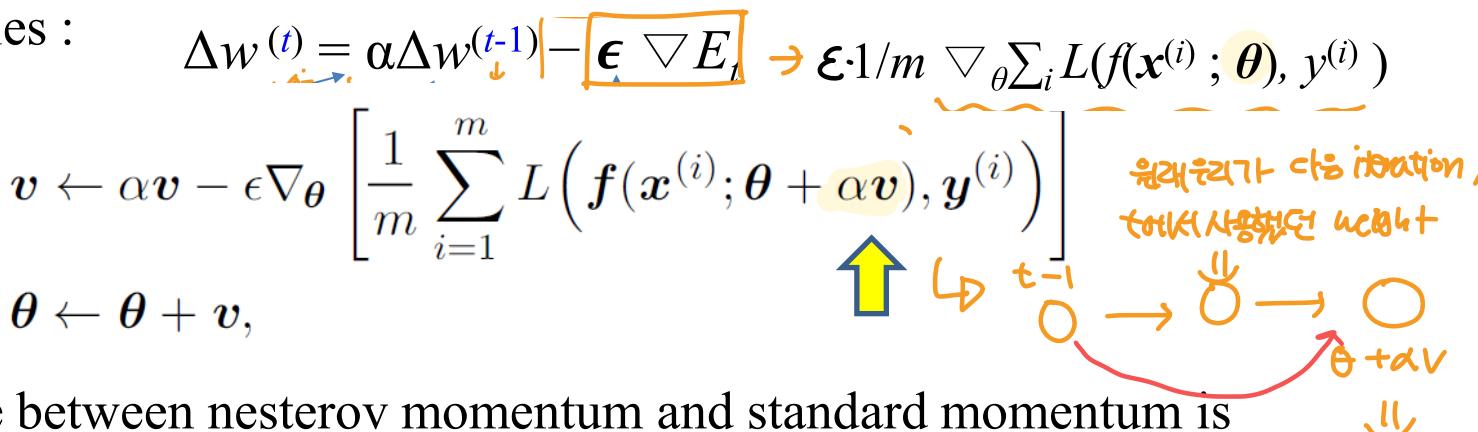


The effect of momentum



이전 iteration의 경사에 대한 영향력
같은 방향으로 가는 것에는 가속도를 더해주고, (가속)
반대 방향으로 가는 것에는 가속도를 빼줌. (감속)

Nesterov Momentum (Sutskever *et al.* 2013)

- A variant of the momentum algorithm
- Inspired by Nesterov's accelerated gradient method (Nesterov, 1983, 2004).
- The update rules :
$$\Delta w^{(t)} = \alpha \Delta w^{(t-1)} - \epsilon \nabla E_t \rightarrow \epsilon \cdot 1/m \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$$
$$v \leftarrow \alpha v - \epsilon \nabla_{\theta} \left[\frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}; \theta + \alpha v), y^{(i)}) \right]$$
$$\theta \leftarrow \theta + v,$$


한국어 설명
• The difference between nesterov momentum and standard momentum is
• Where the **gradient** is evaluated.
With nesterov momentum the gradient is evaluated
• After the current **velocity** is applied.
Thus one can interpret nesterov momentum as attempting to add a **correction factor** to the standard method of momentum.
The complete nesterov momentum algorithm is presented in algorithm 8.3.

Algorithm 8.3 Nesterov momentum

Require: Learning rate ϵ , momentum parameter α .

Require: Initial parameter θ , initial velocity v . $V = \Delta w$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{x^{(1)}, \dots, x^{(m)}\}$ with corresponding labels $y^{(i)}$.

 Apply interim update: $\tilde{\theta} \leftarrow \theta + \alpha v \rightarrow$ Nesterov momentum

 Compute gradient (at interim point): $g \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(x^{(i)}; \tilde{\theta}), y^{(i)})$

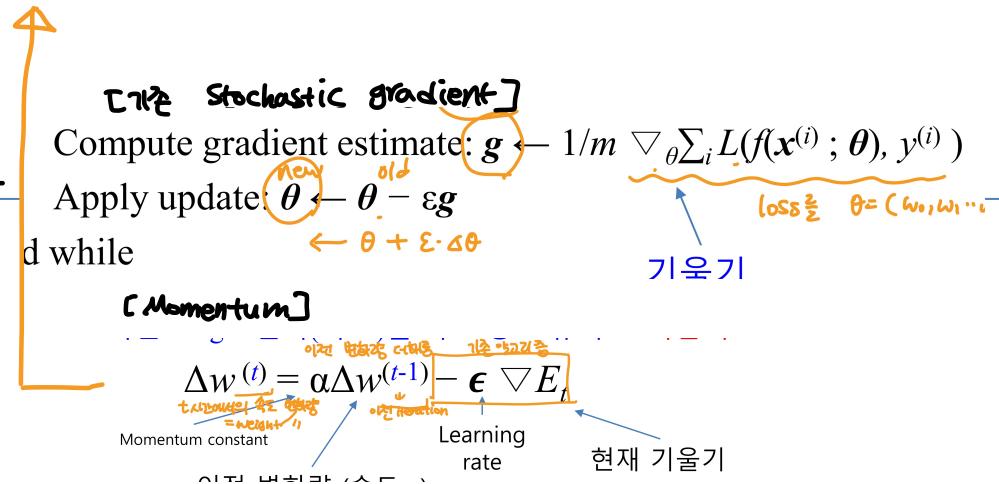
 Compute velocity update: $v \leftarrow \alpha v - \epsilon g \rightarrow$ momentum

 Apply update: $\theta \leftarrow \theta + v$

$$\leftarrow \theta + \alpha v - \epsilon g$$

$$\Leftrightarrow \theta + \alpha \Delta w_{(t-1)} - \epsilon g$$

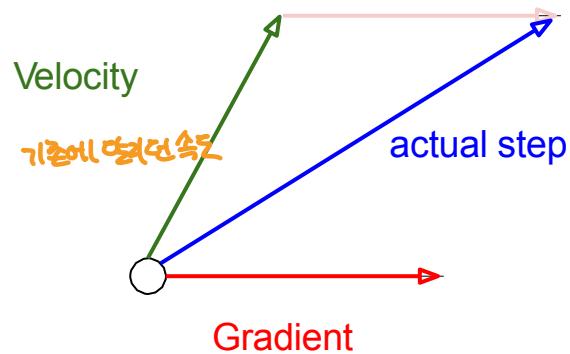
end while



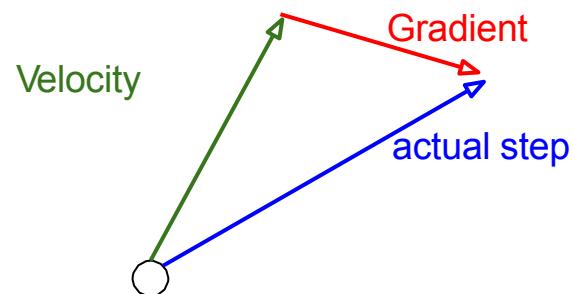
출처 <https://www.deeplearningbook.org/>

Nesterov momentum

Momentum update:



Nesterov Momentum



텐서 플로우 블로그 (Tensor ≈ Blog)

머신러닝(Machine Learning), 딥러닝(Deep Learning) 그리고 텐서(Tensor) 또 파이썬(Python)

Momentum & Nesterov momentum

경사하강법 gradient descent 최적화 알고리즘의 한 종류인 모멘텀 momentum과 네스테로프 모멘텀 nesterov momentum 방식은 여러 신경망 모델에서 널리 사용되고 있습니다. 비교적 이 두가지 알고리즘은 직관적이고 쉽게 이해할 수 있습니다. 이 글에서는 두 알고리즘이 실제 구현에서는 어떻게 적용되는지 살펴 보겠습니다.

모멘텀 알고리즘은 누적된 과거 그래디언트가 지향하고 있는 어떤 방향을 현재 그래디언트에 보정하려는 방식입니다. 일종의 관성 또는 가속도처럼 생각하면 편리합니다. 머신 러닝의 다른 알고리즘들이 그렇듯이 모멘텀 공식도 쓰는 이마다 표기법이 모두 다릅니다. 여기에서는 일리아 서스키비 Illya Sutskever의 페이퍼¹에 있는 표기를 따르겠습니다. 모멘텀 알고리즘의 공식은 아래와 같습니다.

$$\begin{aligned} v_{t+1} &= \mu v_t - \epsilon g(\theta_t) \\ \theta_{t+1} &= \theta_t + v_{t+1} \end{aligned}$$

ϵ 은 학습속도이고 μ 는 모멘트 효과에 대한 가중치입니다. v_t 는 0으로 초기화되어 있고 반복이 될 때마다 현재의 그래디언트 $-\epsilon g(\theta_t)$ 가 다음번 모멘트 v_{t+1} 에 누적됩니다. 그리고 다음번 반복에서 v_{t+1} 가 현재의 모멘트 v_t 로 사용됩니다. 경사하강법에서 모멘트 항이 추가된 것입니다.

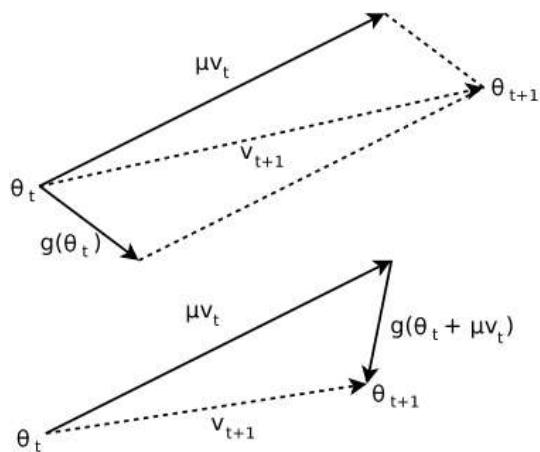


Figure 1. (Top) Classical Momentum (Bottom) Nesterov Accelerated Gradient

이 그림은 일리아 서스키버의 페이퍼¹에서 캡쳐한 것입니다. 위쪽의 그림이 일반 모멘텀 방식을 설명해 주고 있습니다. 파라미터 공간의 현재 위치 θ_t 이전까지 누적된 그래디언트 벡터가 v_t 입니다. 현재 위치의 그래디언트과 구분하기 위해 편의상 이 누적된 그래디언트를 속도라고 부르겠습니다. 속도는 그대로 사용하는 것이 아니고 공식에 표현되어 있듯이 적절한 μ 를 곱합니다. 현재 위치에서의 그래디언트와 속도를 더해 구해진 v_{t+1} 가 θ_t 에 더해지면 θ_{t+1} 가 됩니다. 이런 모멘텀은 그래디언트가 큰 흐름의 방향을 지속하도록 도와 줍니다. 아래 그림은 이런 모멘트의 효과를 그림으로 표현하고 있습니다.²



(a) SGD without momentum



(b) SGD with momentum

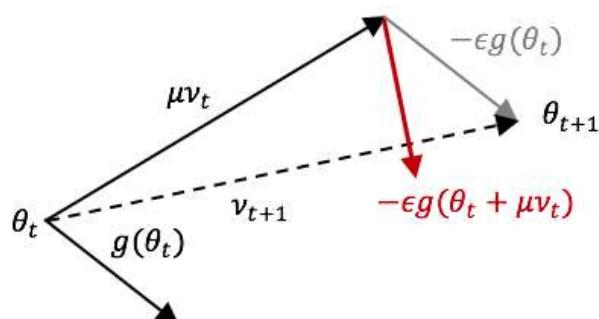
Figure 2: Source: Genevieve B. Orr

그럼 네스테로프 모멘텀은 어떻게 다를까요. Figure 1 의 하단 그림과 아래 공식을 함께 보면 조금 더 이해가 쉽습니다.

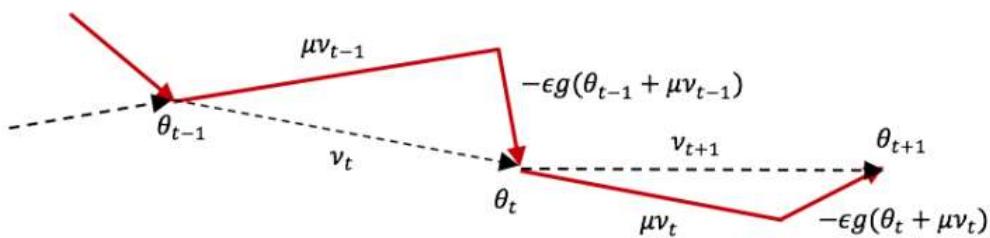
$$\begin{aligned} v_{t+1} &= \mu v_t - \epsilon g(\theta_t + \mu v_t) \\ \theta_{t+1} &= \theta_t + v_{t+1} \end{aligned}$$

이 공식은 모멘텀의 공식과 거의 같습니다. 다만 현재 위치의 그래디언트 $g(\theta_t)$ 를 이용하는 것이 아니고 현재 위치에서 속도 μv_t 만큼 전진한 후의 그래디언트 $g(\theta_t + \mu v_t)$ 를 이용합니다. 사람들은 이를 가리켜 선형적으로 혹은 모형적으로 먼저 진행한 후 에러를 교정한다고 표현합니다. 궁금한 점은 현재 파라미터 위치에서 앞쪽의 그래디언트를 구한다는 것입니다. 반복 루프 안에서 학습이 진행되는 경사하강법의 알고리즘에서 앞선 그래디언트를 어떻게 구하는 것일까요.

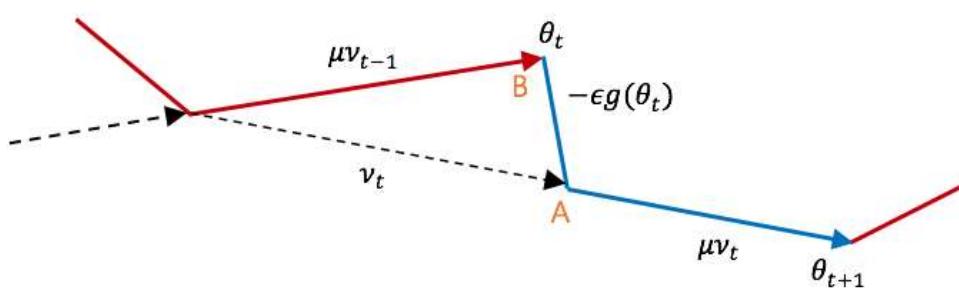
모멘텀 방식과 네스테로프 모멘텀은 확실히 다릅니다. 아래 그림에서 편의상 동일 지점에서 모멘텀과 네스테로프 모멘텀을 한번에 나타내었습니다. 이 그림에서 볼 수 있듯이 μv_t 만큼 이동한 후 현재의 그래디언트 $g(\theta_t)$ 로 진행하는 것과 μv_t 만큼 이동한 지점의 그래디언트 $g(\theta_t + \mu v_t)$ 로 진행하는 것은 다른 결과를 가져다 줍니다.



네스테로프 모멘텀의 진행과정을 조금 더 자세히 보기 위해 반복 스텝 t 의 전후를 이어서 그려 보겠습니다.



우리가 위에서 본 네스테로프 모멘텀 공식으로 현재 θ_{t-1} 에서 θ_t 로 이동하기 위한 v_t 를 구하면 $v_t = \mu v_{t-1} - \epsilon g(\theta_{t-1} + \mu v_{t-1})$ 입니다. 그리고 v_t 는 다음번 반복에서 μv_t 가 되어 새로운 모멘텀 항에 사용됩니다. 앞서 이야기한 것처럼 θ_{t-1} 에서 $g(\theta_{t-1} + \mu v_{t-1})$ 그래디언트를 구하기 위해 코드를 구성하려면 웬지 코드가 복잡해 질 것 같습니다. 그래서 네스테로프를 구현한 여러 코드에서는 약간의 트릭을 사용합니다. 위 그림에서 θ_t 의 위치를 한걸음 뒤로 물려 보겠습니다.



위 그림에서 θ_t 와 θ_{t+1} 에 집중하기 위해 다른 표시들은 삭제했습니다. θ_t 가 A 지점에서 B 지점으로 옮겨졌습니다. 이에 따라 $g(\theta_{t-1} + \mu v_{t-1})$ 가 간단하게 $g(\theta_t)$ 로 바뀌었습니다. θ_t 를 옮기기 전과 후의 값은 확실히 다릅니다. 하지만 네스테로프 모멘텀의 경로를 그대로 따르고 있으므로 학습을 많이 반복하여 최적점에 수렴하면 전체 파라미터 공간에서 A 지점과 B 지점은 거의 동일하게 될 것입니다.

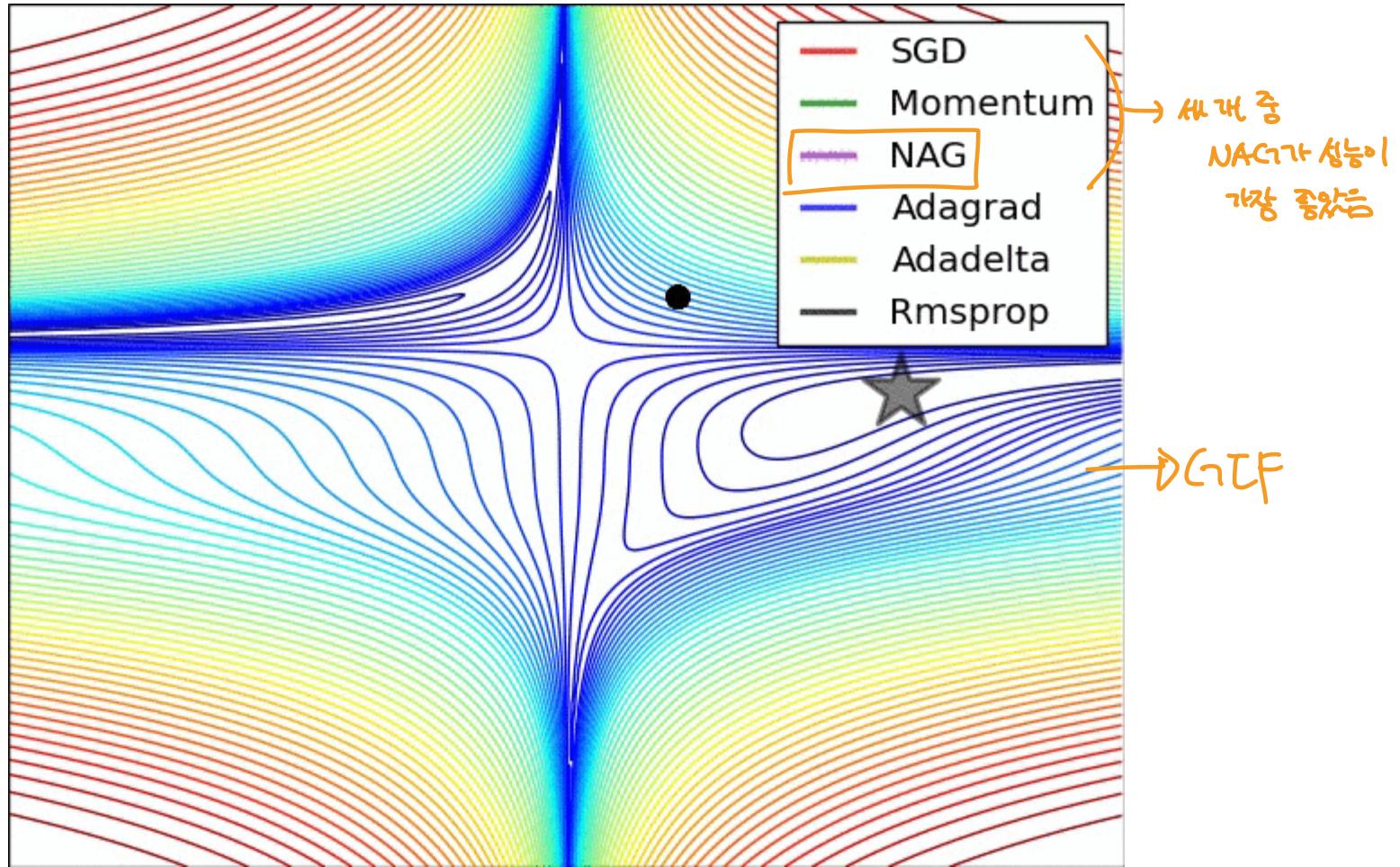
현재 위치에서 $g(\theta_t)$ 는 얻을 수 있고 필요한 것은 μv_t 입니다. 위 그림에서 볼 수 있듯이 $v_t = \mu v_{t-1} - \epsilon g(\theta_t)$ 입니다. 따라서 아래와 같이 쓸 수 있습니다.

$$\mu v_t = \mu(\mu v_{t-1} - \epsilon g(\theta_t))$$

θ_t 에서 $-\epsilon g(\theta_t)$ 만큼 이동하고 다음에 μv_t 를 진행하는 전체 식을 다시 살펴 보면 다음과 같습니다.

$$\begin{aligned}\mu v_t &= \mu(\mu v_{t-1} - \epsilon g(\theta_t)) \\ \theta_{t+1} &= \theta_t - \epsilon g(\theta_t) + \mu v_t = \theta_t - \epsilon g(\theta_t) + \mu(\mu v_{t-1} - \epsilon g(\theta_t))\end{aligned}$$

SGD optimization on loss surface contours



<http://ruder.io/optimizing-gradient-descent/>

AdaGrad = Adaptive Gradient

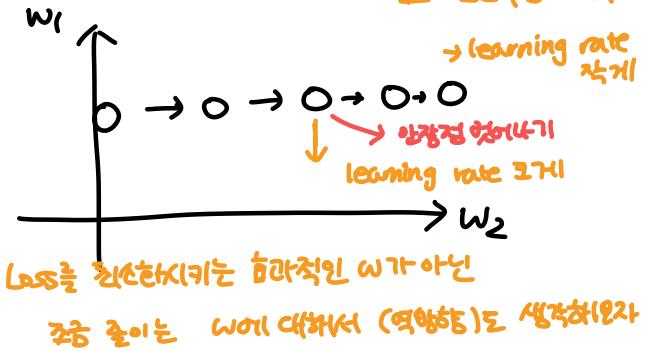
↳ 안장에서의 문제 해결: 한 방향 막고 역방향도 생각하자.

- 자주 나타나는 기울기의 성분보다 드물게 나타나는 기울기 성분을 더 중시해서 파라미터를 업데이트
- 자주 나오는 파라미터 → learning rate 적게
 - 드물게 나오는 파라미터는 크게
- 오차함수의 기울기: $\mathbf{g}_t \equiv \nabla E_t$ $\rightarrow \mathbf{g} = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots)$
- $g_{t,i}$: 기울기 함수의 벡터 성분 $\frac{\partial E}{\partial w_i} \dots$
- 업데이트 양의 성분:

$$-\frac{\epsilon}{\sqrt{\sum_{t'=1}^t g_{t',i}^2}} g_{t,i}$$

iteration 처음부터 현재까지 ($g_i = \frac{\partial E}{\partial w_i}$)² 이므로

시간이 지날수록 learning rate $\frac{\epsilon}{\sqrt{\sum}}$ 는 작아짐.



ϵ : learning rate

The AdaGrad algorithm

Global learning rate ε

Initial parameter θ

Small constant δ , (10^{-7})

Initialize gradient accumulation variable $r = 0$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set

$\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.

 Compute gradient: $g \leftarrow 1/m \nabla_{\theta_i} L(f(x^{(i)}; \theta), y^{(i)})$

Accumulate squared gradient: $r \leftarrow r + g \odot g$

Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$ ↳ g 의 각 원소에 제곱을 한다.

Apply update: $\theta \leftarrow \theta + \Delta\theta$ ↳ $\Delta\theta$ = $[\Delta\omega_0, \Delta\omega_1, \dots, \Delta\omega_N]$

end while

(Division and square root applied element-wise)

$$\mathbf{x}^{(i)} = [x_0, x_1, \dots, x_D]$$

$$\boldsymbol{\theta} = [w_0, w_1, \dots, w_N]$$

$$\Delta\boldsymbol{\theta} = [\Delta w_0, \Delta w_1, \dots, \Delta w_N]$$

$$r = [r_0, r_1, \dots, r_N]$$

$f(x^{(i)}; \theta)$ = network output

$L(f(x^{(i)}; \theta), y^{(i)})$: Loss

$$\begin{aligned} g &\leftarrow 1/m \nabla_{\theta} L = \\ &[\partial L / \partial w_0, \dots, \partial L / \partial w_1, \dots, \partial L / \partial w_N] \end{aligned}$$

○: 벡터의 성분별 곱

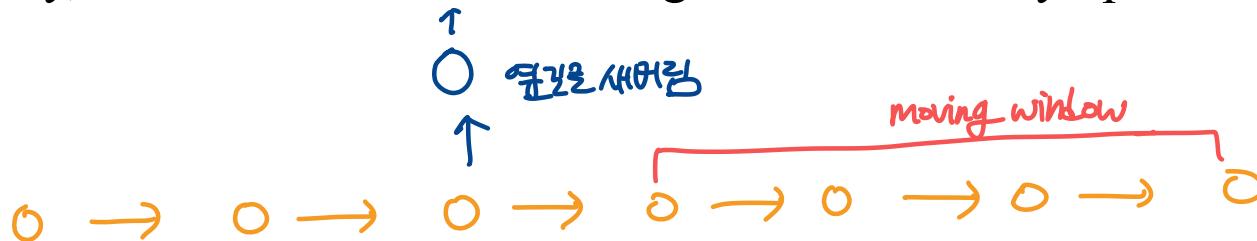
$$r \leftarrow \rho r + (1 - \rho)g \odot g$$

→ rmsprop

→ Adam (adaptive moments)

Adadelta

- Adadelta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients.
- This way, Adadelta continues learning even when many updates have been done.



iteration이 진행될수록 아무리 길어도 학습률은 learning rate이 적으로
그 방향으로 가는게 손속적 \Rightarrow 문제점

\Rightarrow AdaGrad는 처음부터 현재까지 learning rate을 줄여나간다.

Adadelta는 최근 (moving window 내의)의 것들로 learning rate를 줄이자.
과거는 잊고

Rmsprop (Root Mean Square Propagation)

Require: Global learning rate ϵ , decay rate ρ , momentum coefficient α .

Require: Initial parameter θ , initial velocity v .

Initialize accumulation variable $r = 0$

$$\sum_{t'=1}^t g_{t',i}^2 = v$$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{x^{(1)}, \dots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.

 Compute interim update: $\tilde{\theta} \leftarrow \theta + \alpha v$

 Compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(x^{(i)}; \tilde{\theta}), y^{(i)})$

 Accumulate gradient: $r \leftarrow \rho r + (1 - \rho)g \odot g$

 Nesterov momentum

 Compute velocity update: $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot g$. ($\frac{1}{\sqrt{r}}$ applied element-wise)

 Apply update: $\theta \leftarrow \theta + v$

end while

- AdaGrad의 문제: 시간이 흐를수록 update 양이 작아짐.
→ exponentially decaying average to discard history from the extreme past



분류 전체보기 (217) Keras 소스 코드

S/W (160)

1 | keras.optimizers.SGD(lr=0.1, momentum=0.9, nesterov=True)

?

· 네트워크 & 통신 (12)

· 데이터 마이닝 (10)

· 머신러닝 (13)

· 병렬 프로그래밍 (3)

5. Adagrad(Adaptive Gradient, 아다그라드)

· 안드로이드 (23) 아다그라드는 변수의 업데이트 횟수에 따라 학습률(Learning rate)를 조절하는 옵션이 추가된 최적화 방법입니다. 여기서 변

· 알고리즘 (6) 수란 가중치(W) 벡터의 하나의 값(w[i])을 말합니다. 아다그라드는 많이 변화하지 않은 변수들은 학습률(step size)를 크게하

· 웹 프로그래밍 (3)

· 파이썬 (6) 고, 반대로 많이 변화한 변수들에 대해서는 학습률을 적게합니다. 이는 많이 변화한 변수는 최적값에 근접했을 것이라는 가정하

· C & C++ (12) 에 작은 크기로 이동하면서 세밀한 값을 조정하고, 반대로 적게 변화한 변수들은 학습률을 크게하여 빠르게 loss값을 줄입니다.

· JAVA (14)

· OS (29) 아다그라드는 같은 입력 데이터가 여러번 학습되는 학습모델에 유용하게 쓰이는데 대표적으로 언어와 관련된 word2vec이나

· Linux (14) GloVe에 유용합니다. 이는 학습 단어의 등장 확률에 따라 변수의 사용 비율이 확연하게 차이나기 때문에 많이 등장한 단어는 가

· 기타 (15) 중치를 적게 수정하고 적게 등장한 단어는 많이 수정할 수 있기 때문입니다.

H/W (52)

· 아두이노 (10)

· ARTIK (23)

· 프로젝트 (8)

· 기타 (11)

기타 (5)

방명록

수식은 다음과 같습니다.

$$G(t) = G(t-1) + \left(\frac{\partial}{\partial w(t)} Cost(w(t)) \right)^2$$

$$= \sum_{i=0}^t \left(\frac{\partial}{\partial w(i)} Cost(w(i)) \right)^2$$

$$W(t+1) = W(t) - \alpha * \frac{1}{\sqrt{G(t)+\epsilon}} * \frac{\partial}{\partial w(i)} Cost(w(i))$$

$G(t)$ 의 수식을 보면 현재 gradient의 제곱에 $G(t-1)$ 값이 더해집니다. 이는 각 step의 모든 gradient에 대한 sum of squares라는 것을 알 수 있습니다.

$W(t+1)$ 을 구하는 식에서 $G(t)$ 는 ϵ (입실론)값과 더해진 후 루트가 적용되고 α (알파)에 나누어 집니다. 여기서 ϵ (입실론)은 아주 작은 상수를 의미하며, 0으로 나누는 것을 방지해 줍니다. 그리고 α (알파)는 Learning Rate를 나타내며 $G(t)$ 의 크기에 따라 값이 변합니다.

파이썬 소스 코드

1 | g += gradient**2
2 | weight[i] += - learning_rate (gradient / (np.sqrt(g) + e))

?

Tensorflow 소스 코드

1 | optimizer = tf.train.AdagradOptimizer(learning_rate=0.01).minimize(loss)

?

Keras 소스 코드

1 | keras.optimizers.Adagrad(lr=0.01, epsilon=1e-6)

[내용]

6. RMSprop(알엠에스프롭)

RMSprop(알엠에스프롭)은 아다그라드의 $G(t)$ 의 값이 무한히 커지는 것을 방지하고자 제안된 방법으로, 논문과 같은 형태로 발표된 다른 방법들과 달리 제프리 힌튼 교수와 제자들이 코세라(Coursera) 수업에서 소개하였습니다. 링크를 클릭하시면 영상 페이지로 들어가실수 있으며 이론과 수식 도출에 대해 설명하고 있습니다. RMSprop은 지수 이동평균을 이용한 방법입니다.

우선 지수 이동평균에 대해 알아보겠습니다. 지수 이동평균이란 쉽게 말해 최근 값을 더 잘 반영하기 위해 최근 값에 값과 이전 값에 각각 가중치를 주어 계산하는 방법입니다.

$$x_t = \alpha p_t + (1-\alpha)x_{t-1} \quad \text{where} \quad \alpha = \frac{2}{N+1}$$



분류 전체보기 (217)

S/W (160)

- 네트워크 & 통신 (12)
- 데이터 마이닝 (10)
- 머신러닝 (13)
- 병렬 프로그래밍 (3)
- 안드로이드 (23)
- 알고리즘 (6)
- 웹 프로그래밍 (3)
- 파이썬 (6)
- C & C++ (12)
- JAVA (14)
- OS (29)
- Linux (14)
- 기타 (15)

H/W (52)

- 아두이노 (10)
- ARTIK (23)
- 프로젝트 (8)
- 기타 (11)

기타 (5)

방명록

$$\begin{aligned}
 x_k &= \alpha p_k + (1-\alpha)x_{k-1} \\
 &= \alpha p_k + (1-\alpha)\{\alpha p_{k-1} + (1-\alpha)x_{k-2}\} \\
 &= \alpha p_k + \alpha(1-\alpha)p_{k-1} + (1-\alpha)^2 x_{k-2} \\
 &= \alpha p_k + \alpha(1-\alpha)p_{k-1} + (1-\alpha)^2\{\alpha p_{k-2} + (1-\alpha)x_{k-3}\} \\
 &= \alpha p_k + \alpha(1-\alpha)p_{k-1} + \alpha(1-\alpha)^2 p_{k-2} + (1-\alpha)^3 x_{k-3} \\
 &= \dots \\
 &= \alpha(p_k + (1-\alpha)p_{k-1} + (1-\alpha)^2 p_{k-2} + \dots + (1-\alpha)^{N-1} p_{k-N+1}) + (1-\alpha)^N x_{k-N} \\
 &= \alpha \sum_{i=0}^{N-1} (1-\alpha)^i p_{k-i} + (1-\alpha)^N x_{k-N}
 \end{aligned}$$

계산식을 풀어써 보면 위와 같은데 식을 보면 알 수 있듯이 1주기가 지날 때마다 $(1-\alpha)$ 라는 가중치가 이전 값에 곱해지는데, $(1-\alpha)$ 값이 1보다 작기 때문에 시간이 지날수록 영향력이 줄어드는 효과를 볼 수 있습니다. 참고로 필터이론에서는 이런 가중치를 forgetting factor 또는 decaying factor라고 합니다.

RMSprop 수식은 다음과 같습니다.

$$G(t) = \gamma G(t-1) + (1-\gamma) \left(\frac{\partial}{\partial w(i)} Cost(w(i)) \right)^2$$

$$W(t+1) = W(t) - \alpha * \frac{1}{\sqrt{G(t)+\epsilon}} * \frac{\partial}{\partial w(i)} Cost(w(i))$$

기존 Adagrad에서는 $G(t)$ 를 구성하는 두 항이 그냥 더 해지지만 RMSprop에서는 지수평균으로 더해집니다.

파이썬 소스 코드

```

1 | g = gamma * g + (1 - gamma) * gradient**2
2 | weight[i] += -learning_rate * gradient / (np.sqrt(g) + epsilon)

```

Tensorflow 소스 코드

```
1 | optimize = tf.train.RMSPropOptimizer(learning_rate=0.01, decay=0.9, momentum=0.0, epsilon=1e-10).minimize(...)
```

Keras 소스 코드

```

1 | keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
2 | #蒿이퍼-파라메터인 Rho는 γ(감마)를 뜻합니다.
3 | #Rho is a hyper-parameter which attenuates the influence of past gradient.

```

8. Adam(Adaptive Moment Estimation, 아담)

Adam은 Momentum과 RMSprop를 합친 경사하강법입니다. RMSprop의 특징인 gradient의 제곱을 지수평균한 값을 사용하며 Momentum의 특징으로 gradient를 제곱하지 않은 값을 사용하여 지수평균을 구하고 수식에 활용합니다.

수식은 다음과 같습니다.

$$\begin{aligned}
 M(t) &= \beta_1 M(t-1) + (1-\beta_1) \frac{\partial}{\partial w(t)} Cost(w(t)) \\
 V(t) &= \beta_2 V(t-1) + (1-\beta_2) \left(\frac{\partial}{\partial w(i)} Cost(w(i)) \right)^2 \\
 \hat{M}(t) &= \frac{M(t)}{1-\beta_1^t} \quad \hat{V}(t) = \frac{V(t)}{1-\beta_2^t} \\
 W(t+1) &= W(t) - \alpha * \frac{\hat{M}(t)}{\sqrt{\hat{V}(t)+\epsilon}}
 \end{aligned}$$

The Adam algorithm

Adaptive moment
→ momentum + RMSprop

Require: Step size ϵ (Suggested default: 0.001)

Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1]$.
(Suggested defaults: 0.9 and 0.999 respectively)

Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})

Require: Initial parameters θ

↳ V (속도) 라고 생각하면 됨

Initialize 1st and 2nd moment variables $s = 0, r = 0$

Initialize time step $t = 0$

↳ adaptive $\sum g_i \epsilon^2$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{x^{(1)}, \dots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.

 Compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$ → Nesterov X. 훈련 경로

$t \leftarrow t + 1$

 Compute velocity update: $v \leftarrow \alpha v - \epsilon g$ → momentum
 Apply update: $\theta \leftarrow \theta + v$

 Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1)g$

 Update biased second moment estimate: $r \leftarrow \rho_2 r + (1 - \rho_2)g \odot g$

 Correct bias in first moment: $\hat{s} \leftarrow \frac{s}{1 - \rho_1^t}$

 Correct bias in second moment: $\hat{r} \leftarrow \frac{r}{1 - \rho_2^t}$

 Compute update: $\Delta\theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$ (operations applied element-wise) ↳ $\alpha v - \frac{\epsilon}{\sqrt{v}} \odot g$

 Apply update: $\theta \leftarrow \theta + \Delta\theta$

 ↳ $\frac{\text{부자: momentum}}{\text{부여: adaptive}}$

$t=1 \text{ 일 때}$
 $s = \frac{0.5 + (1-0.9)s}{(1-0.9)^2} = g$

 t가 커지면 차이 X, 초반에 g가 적어지는 것을
 방지하기 위해서

RMSprop

end while

Adam 은 이전 글인 [Momentum, AdaGrad 설명](#)에서 언급한 Momentum 과 AdaGrad 를 융합한 방법이다.

[CS 구독하기](#)

Algorithm 1: Adam, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

```

Require:  $\alpha$ : Stepsize
Require:  $\beta_1, \beta_2 \in [0, 1]$ : Exponential decay rates for the moment estimates
Require:  $f(\theta)$ : Stochastic objective function with parameters  $\theta$ 
Require:  $\theta_0$ : Initial parameter vector
 $m_0 \leftarrow 0$  (Initialize 1st moment vector)
 $v_0 \leftarrow 0$  (Initialize 2nd moment vector)
 $t \leftarrow 0$  (Initialize timestep)
while  $\theta_t$  not converged do
     $t \leftarrow t + 1$ 
     $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )
     $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  (Update biased first moment estimate)
     $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$  (Update biased second raw moment estimate)
     $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)
     $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)
     $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)
end while
return  $\theta_t$  (Resulting parameters)

```

출처: <https://arxiv.org/pdf/1412.6980.pdf>

이전 글에서 Momentum 은 새로운 계수로 v 를, AdaGrad 는 h 를 추가하여 최적화를 진행하였는데, Adam 은 두 기법에서 v , h 가 각각 최초 0으로 설정되어 학습 초반에 0으로 biased 되는 문제를 해결하기 위해 고안한 방법이다.

Adam 의 의사코드를 보면 1차 모멘텀인 m 과 2차 모멘텀인 v 를 이용하여 최적화를 진행하는데, 첫번째 항인 m_1 을 계산하는 과정을 살펴보면 쉽게 이해할 수 있다.

$$\begin{aligned}
 m_1 &\leftarrow \beta_1 m_0 + (1 - \beta_1) g_1 \\
 \widehat{m}_1 &\leftarrow \frac{m_1}{1 - \beta_1^1} = \frac{\beta_1 m_0}{1 - \beta_1^1} + \frac{(1 - \beta_1) g_1}{1 - \beta_1^1} \\
 &= 0 + g_1 (\because m_0 = 0)
 \end{aligned}$$

결론적으로, $m_1 = g_1$ (첫번째 기울기) 이 됨으로써 기존에 $m_0 = 0$ 로 인해서, $m_1 = 0.1g_1 (\because \beta_1 = 0.9)$ 이 되어 0에 biased 됨이 해결되었다.

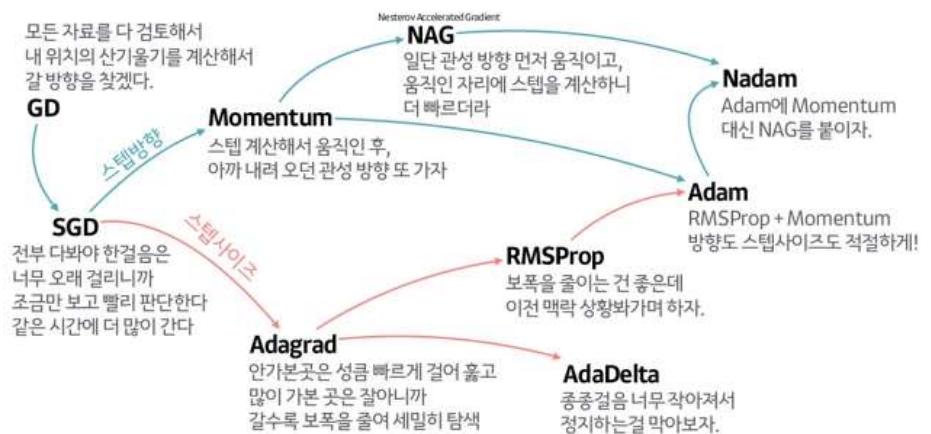
$$\begin{aligned}
 m_2 &\leftarrow \beta_1 m_1 + (1 - \beta_1) g_2 \\
 \widehat{m}_2 &\leftarrow \frac{m_2}{1 - \beta_1^2} = \frac{\beta_1 m_1}{1 - \beta_1^2} + \frac{(1 - \beta_1) g_2}{1 - \beta_1^2} \quad \text{CS 구독하기} \\
 &= 4.73m_1 + 0.52g_2 (\because \beta_1 = 0.9)
 \end{aligned}$$

두번째 항부터는 뒤로 갈 수록 가장 최신 기울기 한 개가 전체 평균에 10% 정도 영향을 주니 가중이동평균 목적에 부합한다.

$$\theta_t \leftarrow \theta_{t-1} - \frac{\alpha \widehat{m}_t}{\sqrt{\widehat{v}_t + \epsilon}}$$

알고리즘의 마지막 계산 순서인 위 부분도, v 를 m 으로 나누는 선택을 할 수도 있었는데 그렇게 하지 않은 이유는 m 은 매번 구해지는 기울기에 따라 늘었다 줄었다 하는 값이지만 v 는 제곱이 더해지므로 무조건 커지는 값이기 때문에 학습할 때 step 이 너무 크지 않도록 적당히 작은 값을 움직이도록 하기 위해서 m<v 인 조건을 이용한 것으로 이해된다.

마지막으로 Optimizer 의 계보(?) 를 설명하는 간단한 그림을 첨부한다.



일반적으로 가장 많이 쓰이는 알고리즘이 Adam.

The Adam algorithm

- Adaptive moments.
- A variant on the combination of rmsprop and momentum.
- Bias correction for the fact that first and second moment estimates start at zero
↳ 초기화 설정
- Adam is generally regarded as being fairly robust to the choice of hyperparameters.

↳ 여러개의 모수들 선정이 번거롭게 영향을 많이 주는데

Adam은 아주 선정이 좀 이상해도 성능이 좋다.

Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

$t \leftarrow t + 1$

Compute velocity update: $v \leftarrow \alpha v - \epsilon \mathbf{g}$ → momentum

Apply update: $\theta \leftarrow \theta + v$

d while

Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1) \mathbf{g}$ →

Update biased second moment estimate: $r \leftarrow \rho_2 r + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$

$$s_1 \\ s_2 = \rho_1 s_1 + (1 - \rho_1) \mathbf{g}_1$$

$$s_3 = \rho_1 s_2 + (1 - \rho_1) \mathbf{g}_2 \\ = \rho_1^2 s_1 + \rho_1(1 - \rho_1) \mathbf{g}_2 + (1 - \rho_1)^2 \mathbf{g}_3 \\ s_4 = \rho_1 s_3 + (1 - \rho_1) \mathbf{g}_4 \\ = \rho_1^3 s_1 + \rho_1^2(1 - \rho_1) \mathbf{g}_2 + \rho_1(1 - \rho_1)^2 \mathbf{g}_3 \\ + (1 - \rho_1)^3 \mathbf{g}_4$$