Improving Deep Neural Networks

Andrew Ng

머신러닝 어플리케이션 설정하기

신경망을 훈련시킬 때는 많은 결정을 내려야 함

신경망이 몇 개의 층을 가지는지
각각의 층이 몇 개의 은닉 유닛을 가지는지
학습률
서로 다른 층에 사용하는 활성화 함수는 무엇인지

처음부터 정확한 파라미터를 찾는 것은 불가능하다
실질적으로 머신러닝을 적용하는 것은 매우 반복적인 과정

딥러닝을 적용하는 것은 어플리케이션의 네트워크에 대한 좋은 선택을 찾기 위해 이 사이클을 여러 번 돌아야 하는 매우 반복적인 과정

머신러닝 어플리케이션 설정하기
Train/ Dev/ Test sets

1) 이 사이클이 얼마나 효율적으로 돌 수 있는지와 데이터 세트를 잘 설정하는 것

- 훈련, 개발, 테스트세트를 잘 설정하는 것은 과정을 더 효율적으로 만듬
- 전통적인 방법은 모든 데이터를 가져와서 일부를 잘라서 훈련 세트를 만들고 다른 일부는 교차 검증 세트로 만듬

훈련세트에 대해 계속 훈련 알고리즘을 적용시키면서, 개발 세트 혹은 교차 검증 세트에 대해 다양한 모델 중 어떤 모델이 가장 좋은 성능을 내는지 확인한다
이 과정을 충분히 거치고 더 발전시키고 싶은 최종 모델이 나오면 테스트 세트에 그 모델을 적용시켜 알고리즘이 얼마나 작동하는지 평가한다

머신러닝 어플리케이션 설정하기
Train/ Dev/ Test sets

머신러닝 이전 시대는 70/30으로 나누거나 60/20/20으로 나누는 것이 최적의 관행이었다

그러나 총 100만개 이상의 샘플이 있는 현대 빅데이터 시대는

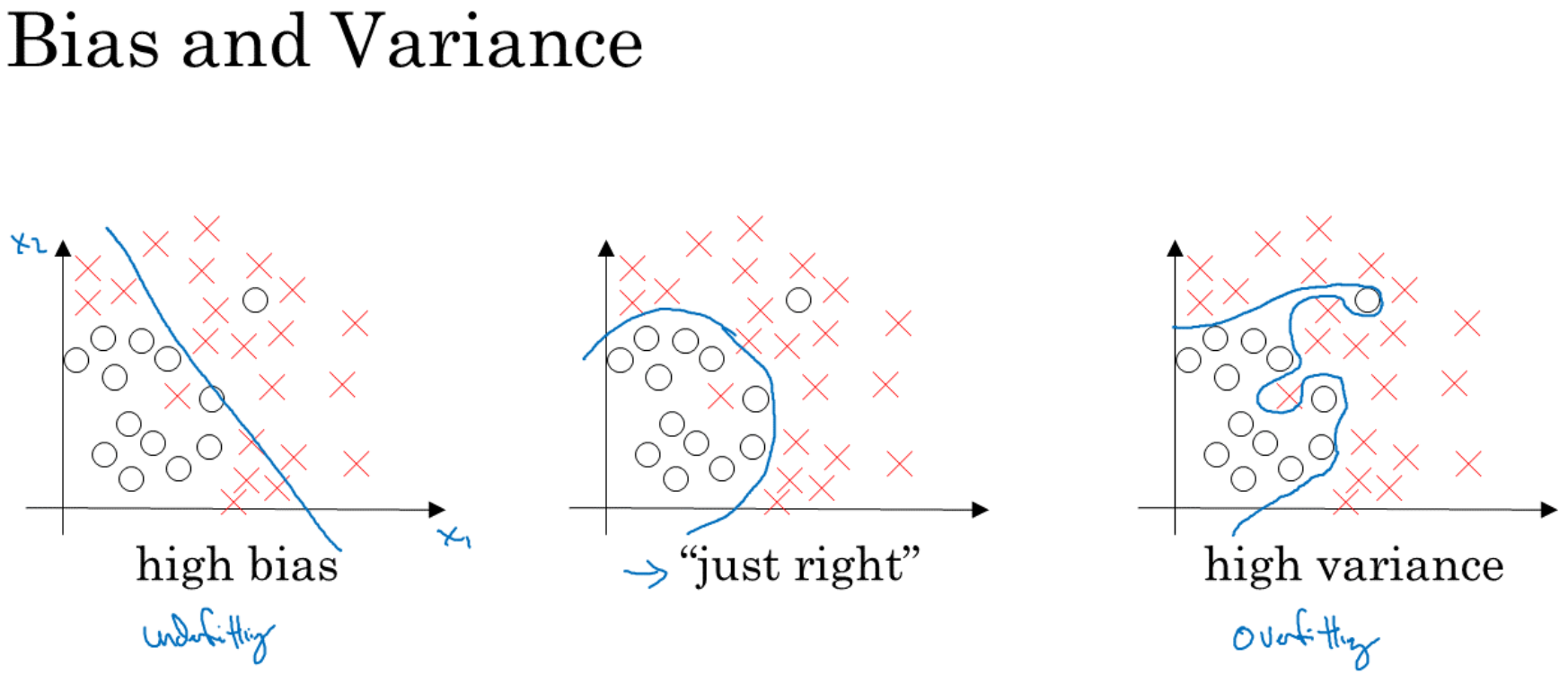
Train 1,000,000 dev 10,000, test 10,000정도로

Dev와 test가 훨씬 더 작은 비율이 되는 것이 트렌드이다 (1%로 줄임)

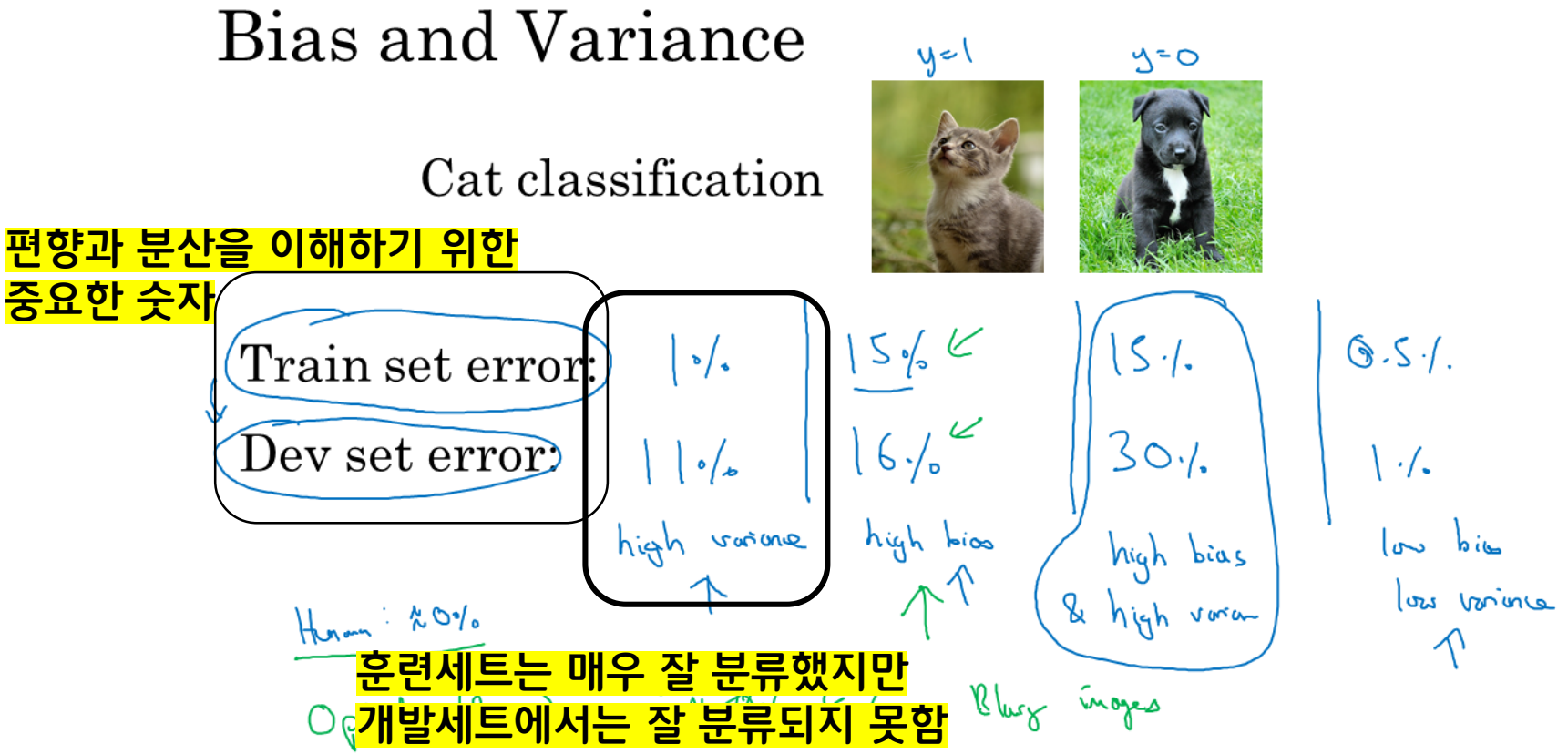
Bias and Variance - Over Fitting

bias variance trade-off : bias와 variance는 반비례 관계라 trade-off가 필요하다.
bias를 낮춰 성능을 올리고!!! variance를 낮춰 over fitting을 막자!!!

딥러닝 시대에 여전히 편향과 분산에 관한 이야기는 하지만 편향과 분산에 대한 트레이드 오프에 관한 이야기는 적어졌다



Bias and Variance - Over Fitting



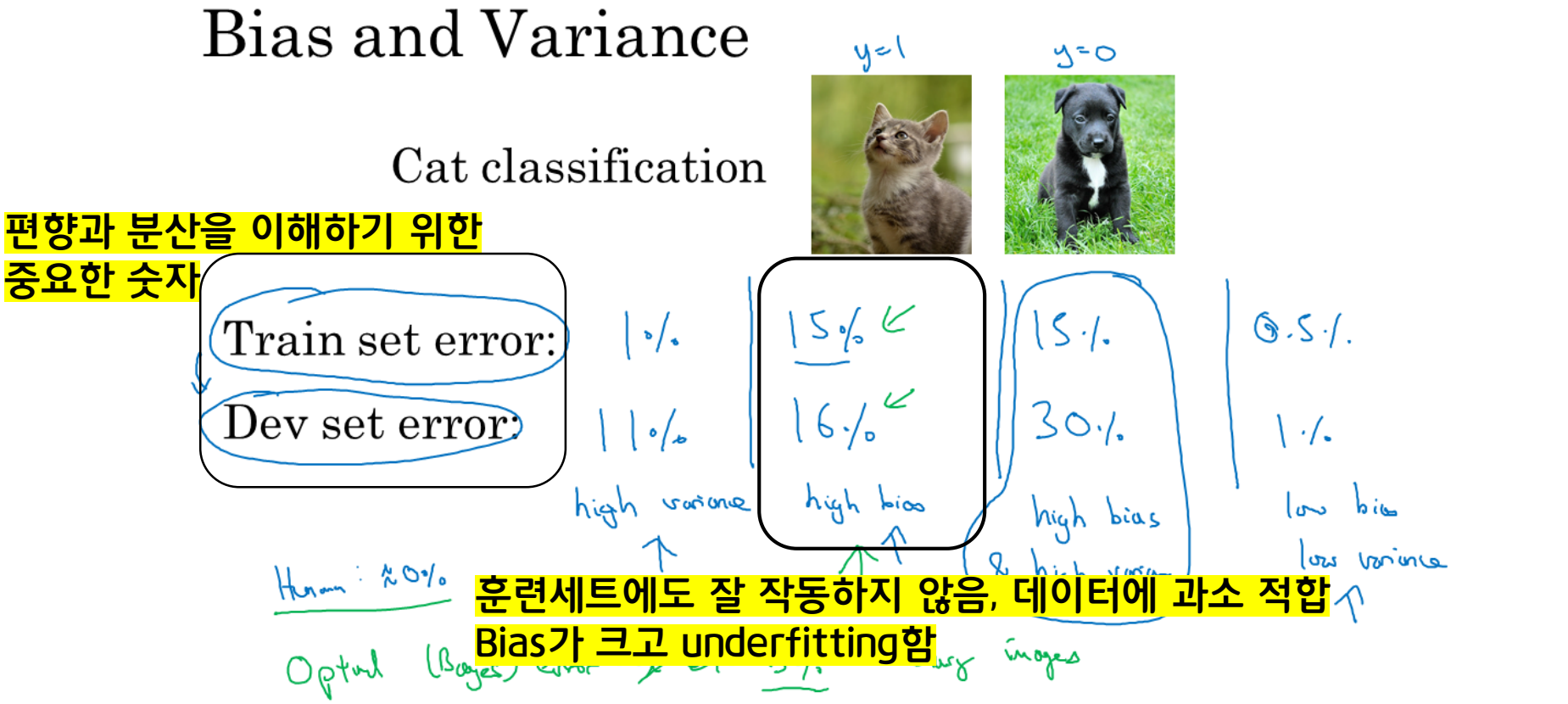
편향과 분산을 이해하기 위한
중요한 숫자

훈련세트는 매우 잘 분류했지만
개발세트에서는 잘 분류되지 못함

즉, 훈련세트에 과대적합 되어서 개발세트에 있는 교차검증세트에서 일반화 되지 못한 경우, train set과 dev set으로 알고리즘이 높은 분산을 갖는다는 것 진단 가능

Andrew Ng

Bias and Variance - Over Fitting



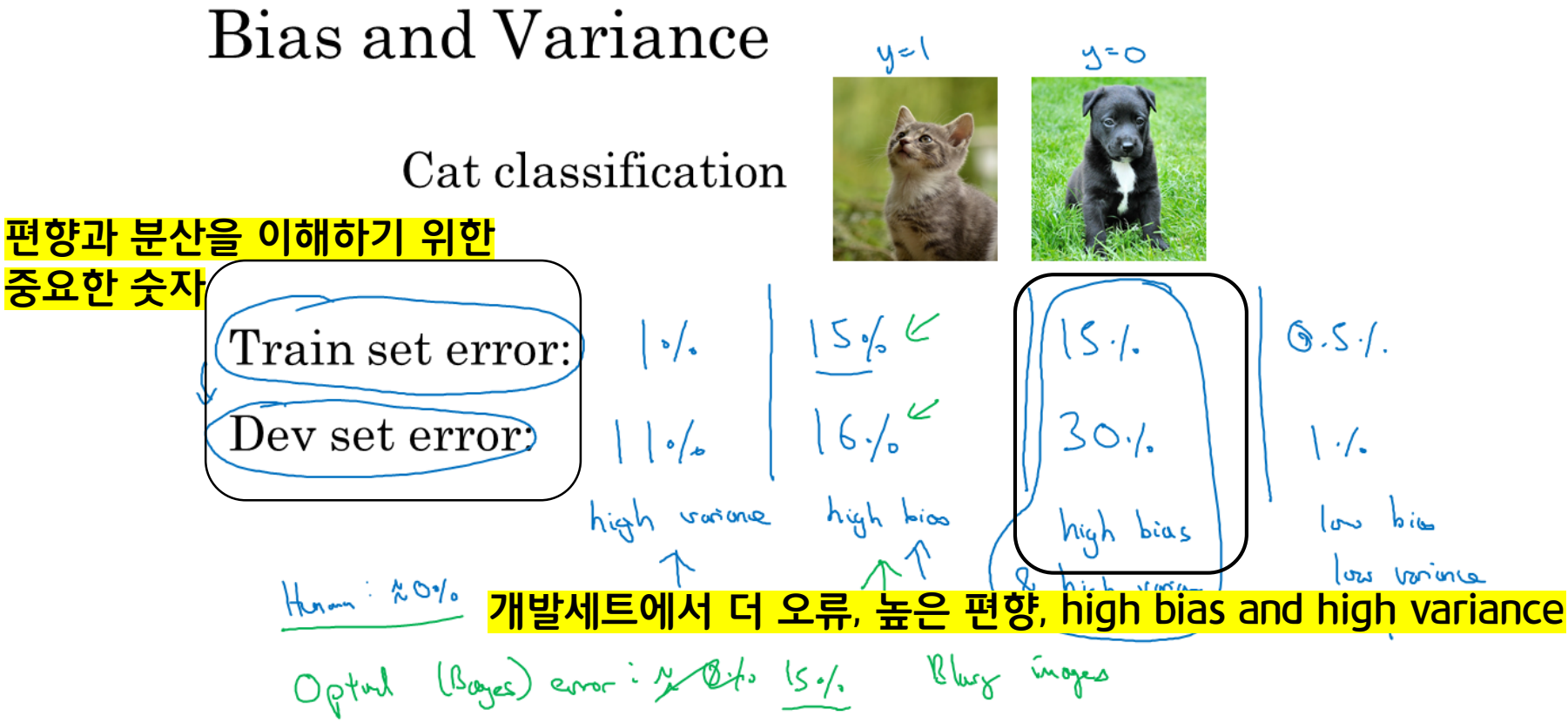
편향과 분산을 이해하기 위한
중요한 숫자

훈련세트에도 잘 작동하지 않음, 데이터에 과소 적합
Bias가 크고 underfitting함

합리적인 수준에서 개발세트에서 일반화 되고 있음
개발세트의 성능이 훈련세트보다 1%밖에 나쁘지 않음, 높은 편향

Andrew Ng

Bias and Variance - Over Fitting



편향과 분산을 이해하기 위한
중요한 숫자

개발세트에서 더 오류, 높은 편향, high bias and high variance

Andrew Ng

Human error와 비교해야한다 (basian error)
어떤 분류에도 잘 작동하지 않을 경우 편향과 분산을 잘 분석해야함

Basic “recipe” for machine learning

최초의 모델을 훈련하고 난 뒤 처음으로 질문하는 것은, 알고리즘이 높은 편향을 가지는지,
높은 편향을 평가하기 위해서는 훈련 세트 혹은 훈련 데이터의 성능을 봐야 한다

높은 현향을 가져서 훈련세트에도 잘 맞지 않는다면 시도할 수 있는 것은
더 많은 은닉층 혹은 은닉 유닛을 갖는 네트워크를 선택하는 것이나 아니면,
더 오랜 시간 훈련시키거나, 다른 발전된 최적화 알고리즘을 사용

다양한 신경망 아키텍처가 있음, 이 문제에 더 잘맞고 잘 작동할 수도 있고 그렇지 않을 수도있음

Basic “recipe” for machine learning

더 큰 네트워크를 갖는 것은 대부분 도움이 되고 더 오래 훈련시키는 것은 도움이 안 될 수도 있지만
해는 되지 않는다, 최소한 편향 문제를 해결할 때까지 이 방법을 시도!

최소한 훈련 세트에 대해서라도 잘 맞게 될 때까지 이 과정을 반복해야 한다
보통 충분히 큰 네트워크라면, 훈련데이터에는 잘 맞출 수도 있다

최소한 인간이 구별할 수 있다면, 즉 베이즈 오차가 그렇게 높지 않다면,
그것보다 더 크게 훈련하는 경우 최소한 훈련 세트에 대해서는 잘 맞을 것이다

Basic “recipe” for machine learning

편향을 수용 가능한 크기로 줄이게 되면,

그 다음에 물어볼 것은 분산에 문제가 있는가?

이를 평가하기 위해 개발 세트 성능을 보게 됨

꽤 좋은 훈련 세트 성능에서 꽤 좋은 개발 세트 성능을 일반화 할 수 있는지

높은 분산 문제가 있을 때 이를 해결하는 가장 좋은 방법은 데이터를 더 얻는 것!

More data(데이터 더 얻기)

Regulization(정규화)

NN achi(다른 신경망 아키텍처를 찾음)

Basic “recipe” for machine learning

높은 편향이냐, 분산이냐에 따라 시도해볼 수 있는 방법이 아주 달라질 수 있음

높은 편향 문제가 있다면, 더 많은 훈련 데이터를 얻는 것은 크게 도움이 되지 않음
따라서 편향과 분산, 혹은 둘 다의 문제가 얼마나 있는지 명확히 하는 것은
가장 유용한 선택을 하게 해 줌

Basic “recipe” for machine learning

Bias-variance trade off

딥러닝 이전 시대에는 틀이 많지 않았음

Bias와 variance 서로 나쁘게 하지 않고 편향만 감소시키거나 분산만 감소시키는 툴이 많이 없었음

현재 딥러닝 빅데이터 시대. 더 큰 네트워크를 훈련시키고 더 많은 데이터를 얻는 것이

대부분 분산을 건드리지 않고 편향만 감소시킨다

최종적으로 잘 정규화 된 네트워크를 갖는 방법도 있음

더 큰 네트워크를 훈련시키는 것은 대부분 절대로 해가 없다!

Regularization

높은 분산으로 신경망이 데이터를 과대적합하는 문제가 의심된다면, 가장 처음으로 시도해야 하는 것은 정규화이다

높은 분산을 해결하는 다른 방법은 더 많은 훈련 데이터를 얻는 것이다

근데 비용이 많이 듦으로 차선책으로 정규화를 한다

Regularization

Logistic regression

$$\min_{w,b} J(w,b)$$

$w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

λ = regularization parameter

lambda

lambd

로지스틱 회귀는 비용함수 J를 최소화 해야한다

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$$

$$+ \frac{\lambda}{2m} b^2$$

람다를 2m으로 나누고 w제곱을 노름으로 곱해준 값 더해줌

로지스틱 회귀에 정규화를 추가하기 위해서

정규화 매개변수라고 부르는 람다를 추가해야함

L_2 regularization $\quad \|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

L_1 regularization $\quad \frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m}\|w\|_1$

Regularization

Logistic regression

$\min_{w,b} J(w,b)$

$w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

λ = regularization parameter

lambda lambd

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$$

L_2 regularization $\quad \|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ ←

L_1 regularization $\quad \frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m}\|w\|_1$

w will be sparse

L2정규화는 매개변수 벡터 w의 유클리드 노름 L2노름을 사용한다

왜 매개변수 w에만 정규화 b는 왜 추가하지 않음. 생략한 것임
꽤 높은 차원의 매개변수 벡터큰 w임, 모든 매개변수는 b가 아닌 w에 있음

Andrew Ng

Regularization

Logistic regression

$\min_{w,b} J(w,b)$

$w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

λ = regularization parameter

lambda

lambd

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$$

+ $\frac{\lambda}{2m} b^2$

omit

L2 regularization $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

L1 regularization $\frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m}\|w\|_1$

w will be sparse

L2노름 대신에 더해줌(L1 정규화)

L2는 가장 일반적인 정규화

W가 희소해지는데 이는, w안에 0이 많아진다는 의미
모델을 압축하겠다는 목표가 있지 않은 이상 Ng
이 정규화를 많이 사용하지 않음

Regularization

Logistic regression

$\min_{w,b} J(w,b)$

$w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

λ = regularization parameter

lambda

lambd

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$$

omit

L_2 regularization $\quad \|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

L_1 regularization $\quad \frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m}\|w\|_1$

w will be sparse

하이퍼파라미터 람다(regularization parameter) 정규화 매개변수
Dev로 다양한 값을 시도해서 훈련세트에 잘 맞으면서
두 매개변수의 노름을 잘 설정해 과대 적합을 막을 수 있는 최적의 값을 찾음

Andrew Ng

Regularization

정규화를 위해 람다를 2m으로 나눈 값 곱하기
매개변수 w노름 제곱의 모든 값을 더해줌

Neural network

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$$

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$$

$w^{[l]} : (n^{[l]}, n^{[l-1]})$

"Frobenius norm"

$\|\cdot\|_2^2$ $\|\cdot\|_F^2$

행렬의 노름은 노름의 제곱을 i와 j에 해당하는 각각의 행렬의 원소를 제곱한 것을 모두 더해준 값
"프로베니우스노름" : 행렬의 원소 제곱의 합이라는 뜻

$$\frac{\partial J}{\partial w^{[l]}} = dw^{[l]}$$

$$w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$$

"Weight decay"

$$w^{[l]} := w^{[l]} - \alpha\left[(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}\right]$$

$$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha\,(\text{from backprop})$$

$$= \underbrace{\left(1 - \frac{\alpha\lambda}{m}\right)}_{<1} w^{[l]} - \alpha\,(\text{from backprop})$$

Andrew Ng

Regularization

Neural network

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$$

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$$

$w^{[l]} : (n^{[l]}, n^{[l-1]})$

정규화 항을 더해주면 이런 식이 됨

$\|\cdot\|_F^2$

$$dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} w^{[l]}$$

$$w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$$

$$\frac{\partial J}{\partial w^{[l]}} = dw^{[l]}$$

"Weight decay"

$$w^{[l]} := w^{[l]} - \alpha\left[(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}\right]$$

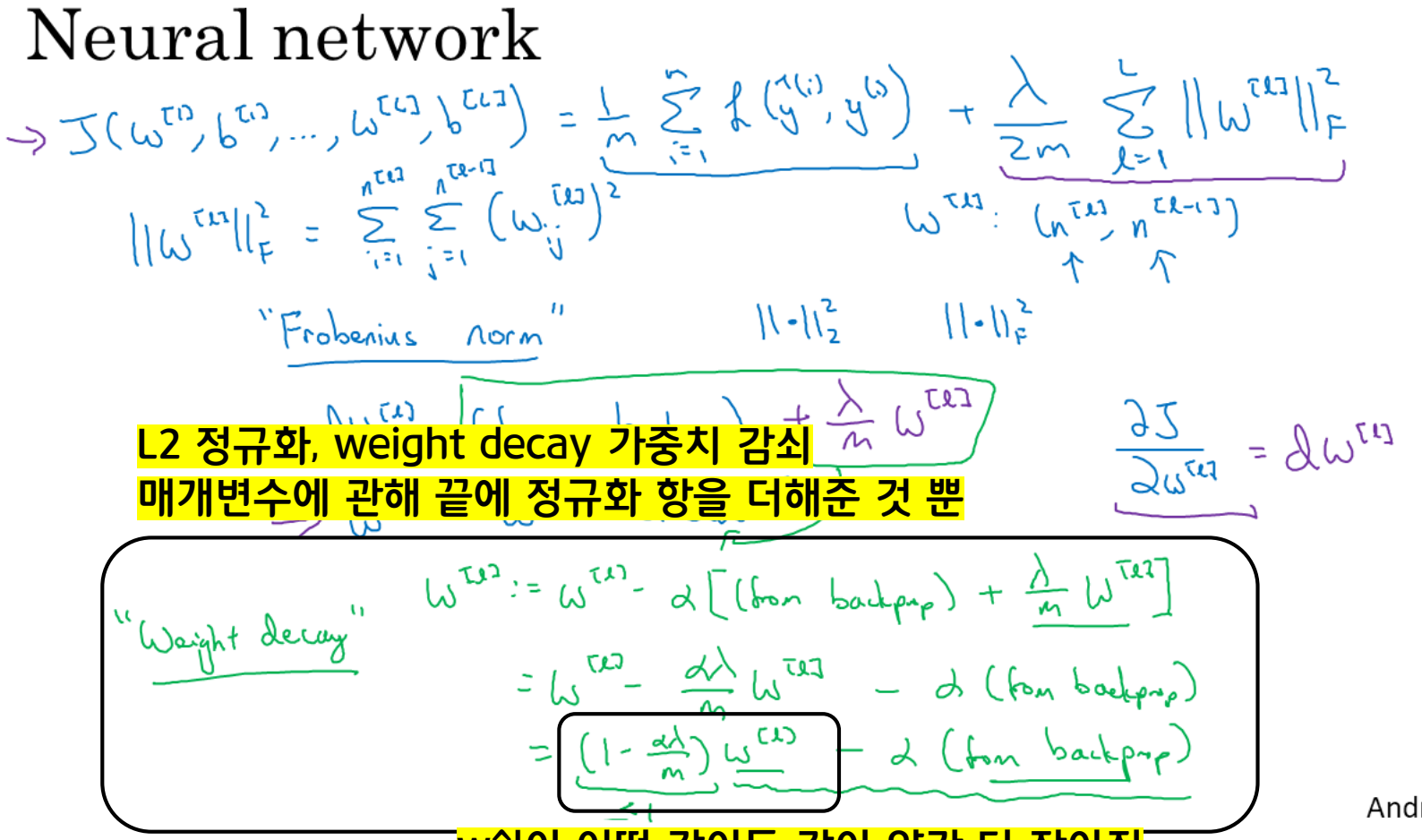
$$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha\,(\text{from backprop})$$

$$= \underbrace{\left(1 - \frac{\alpha\lambda}{m}\right)}_{<1} w^{[l]} - \alpha\,(\text{from backprop})$$

Andrew Ng

Regularization

weight에 제약을 줘서 weight의 값들이 작아지도록 하는 것이다.



L2 정규화, weight decay 가중치 감쇠

매개변수에 관해 끝에 정규화 항을 더해준 것 뿐

Andrew Ng

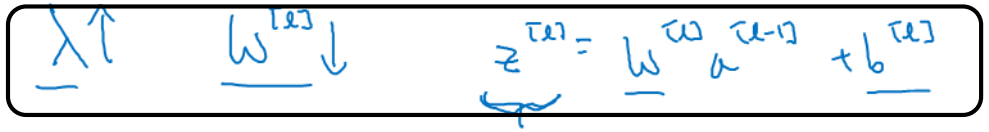
w^l이 어떤 값이든 값이 약간 더 작아짐

W^l에 알파 곱하기 dw^l의 값을 빼서 값을 업데이트 하는 것은 원래 경사하강법과 같지만 w^l에 1 보다 작은 값을 곱해줌

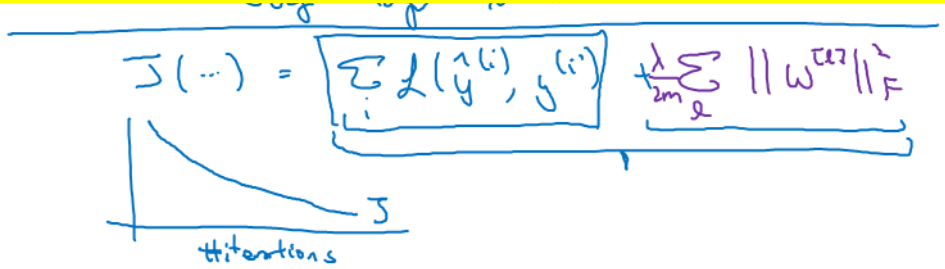
Why Regulatization Reduces overfitting

How does regularization prevent overfitting?

0에 가까우면 선형에 가까워짐



정규화 매개변수인 람다가 커질 때 비용함수가 커지지 않으려면 상대적으로 w가 작아짐



Andrew Ng

W가 작으면 z도 작아짐

g(z) 1차원에 가까워짐, 따라서 모든 함수는 선형회귀처럼 거의 직선 함수를 갖게 됨

러면 전체적으로 신경망이 단순해 지는 효과가 있다. 모델이 단순해진다는 것은 variance가 낮아진다는 것이다.

Dropout Regulatization

매우 강력한 정규화 기법 드롭아웃

과적합한 신경망을 훈련시키는 경우를 생각해보자

드롭아웃은 신경망의 각각의 층에 대해 노드를 삭제하는 확률을 설정하는 것

각각의 층에 대해, 각각의 노드마다 동전을 던지면 0.5의 확률로 해당 노드를 유지하고,

0.5의 확률로 노드를 삭제하게 됨

그 후 삭제된 노드의 들어가는 링크와 나가는 링크 모두 삭제함

더 작고 간소화된 네트워크가 됨

그럼 이 감소화된 네트워크에서 하나의 샘플을 역전파로 훈련시킴

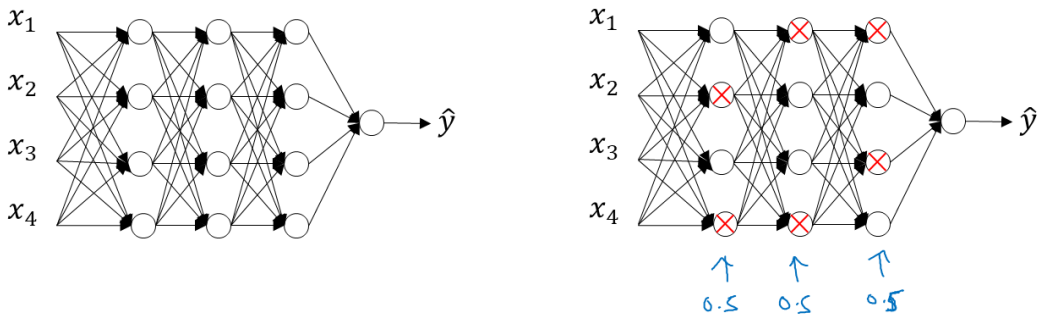
다른 샘플에 대해서도 동전을 던지고 다른 세트들의 노드들을 남기고 다른 세트들의 노드들은 삭제

Dropout Regulatization

노드를 무작위로 삭제하는 기법 → 잘돌아 감

각각의 샘플에서 더 작은 네트워크를 훈련시키는 방식

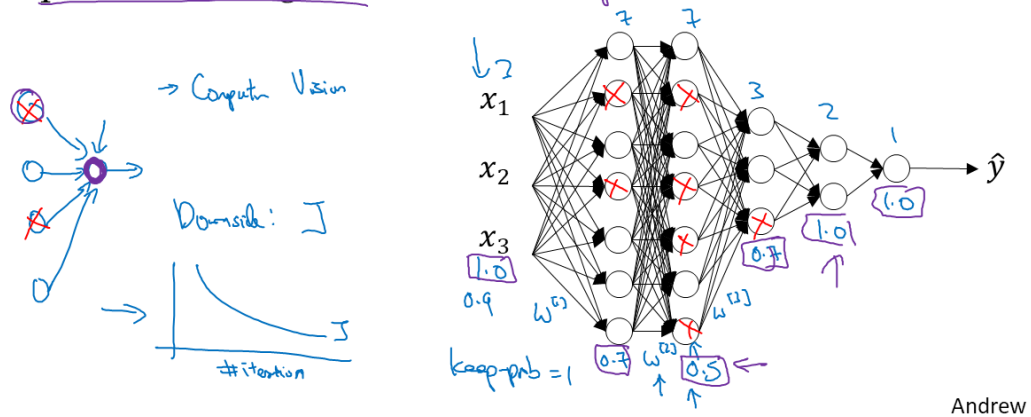
Dropout regularization



Andrew Ng

Why does drop-out work?

Intuition: Can't rely on any one feature, so have to spread out weights. ⇝ Shrink weights.



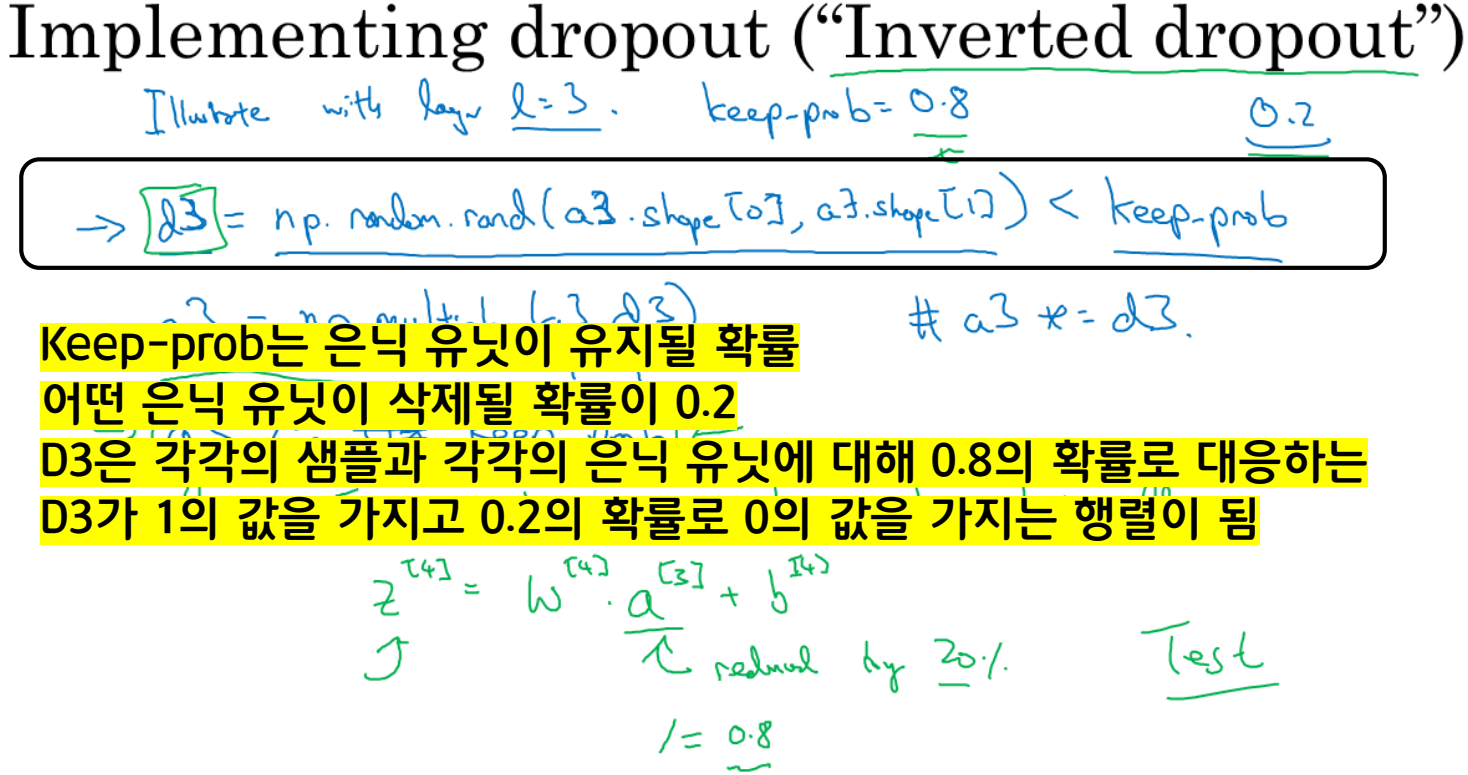
Andrew Ng

weight가 치중되어 있으면 치중되어 있는 unit이 죽었을 때 모델은 큰 영향을 받는다. 따라서 영향을 줄이기 위해 weight를 분산시키게 되고 그에 따라 weight의 norm 줄어들어든다.

Implementing dropout("Inverted dropout") 역드롭아웃

일반적인 기법 역 드롭아웃

층이 3인 경우를 예시로 단일 층에서 드롭아웃을 구현하는 방법

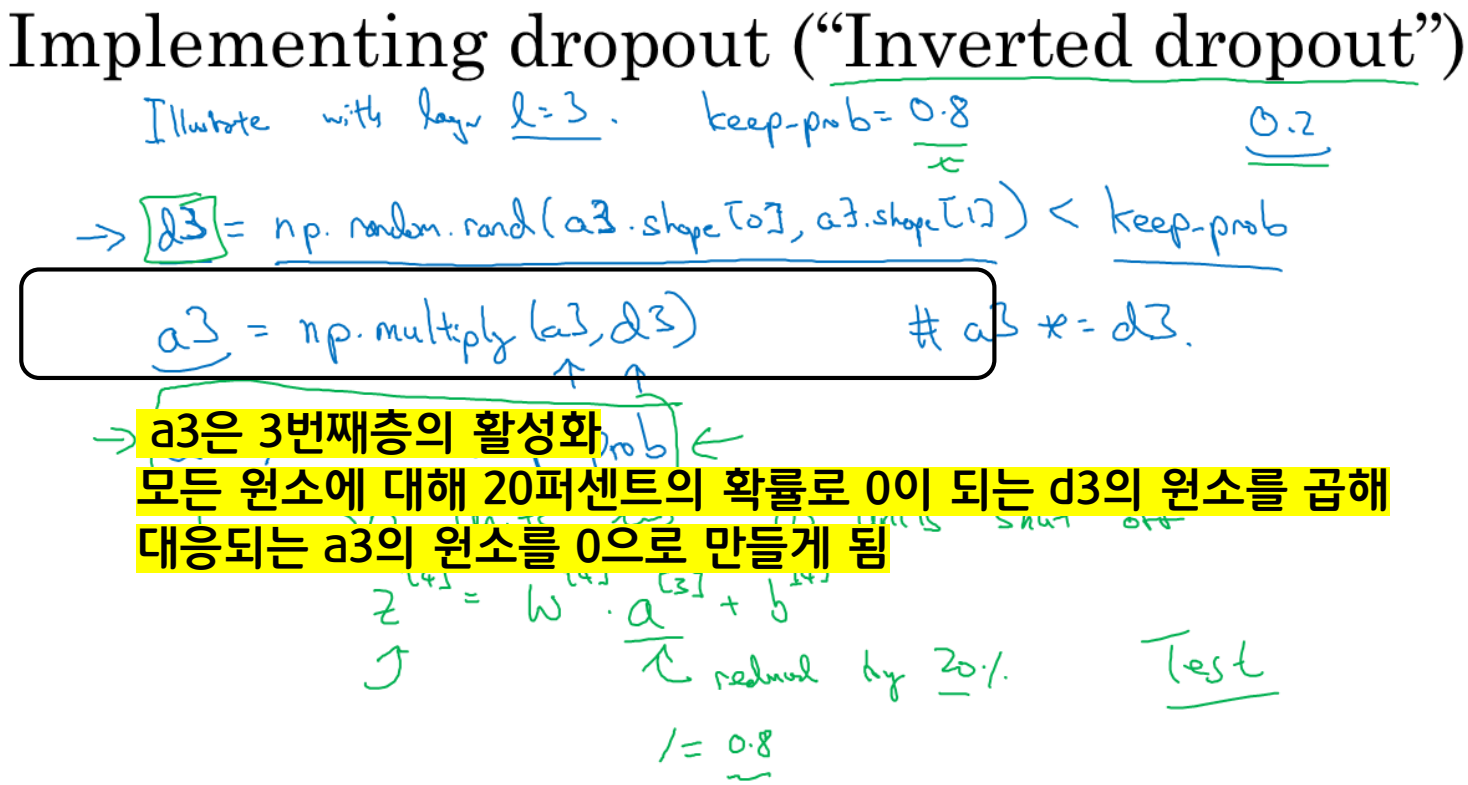


Andrew Ng

Implementing dropout("Inverted dropout") 역드롭아웃

일반적인 기법 역 드롭아웃

층이 3인 경우를 예시로 단일 층에서 드롭아웃을 구현하는 방법

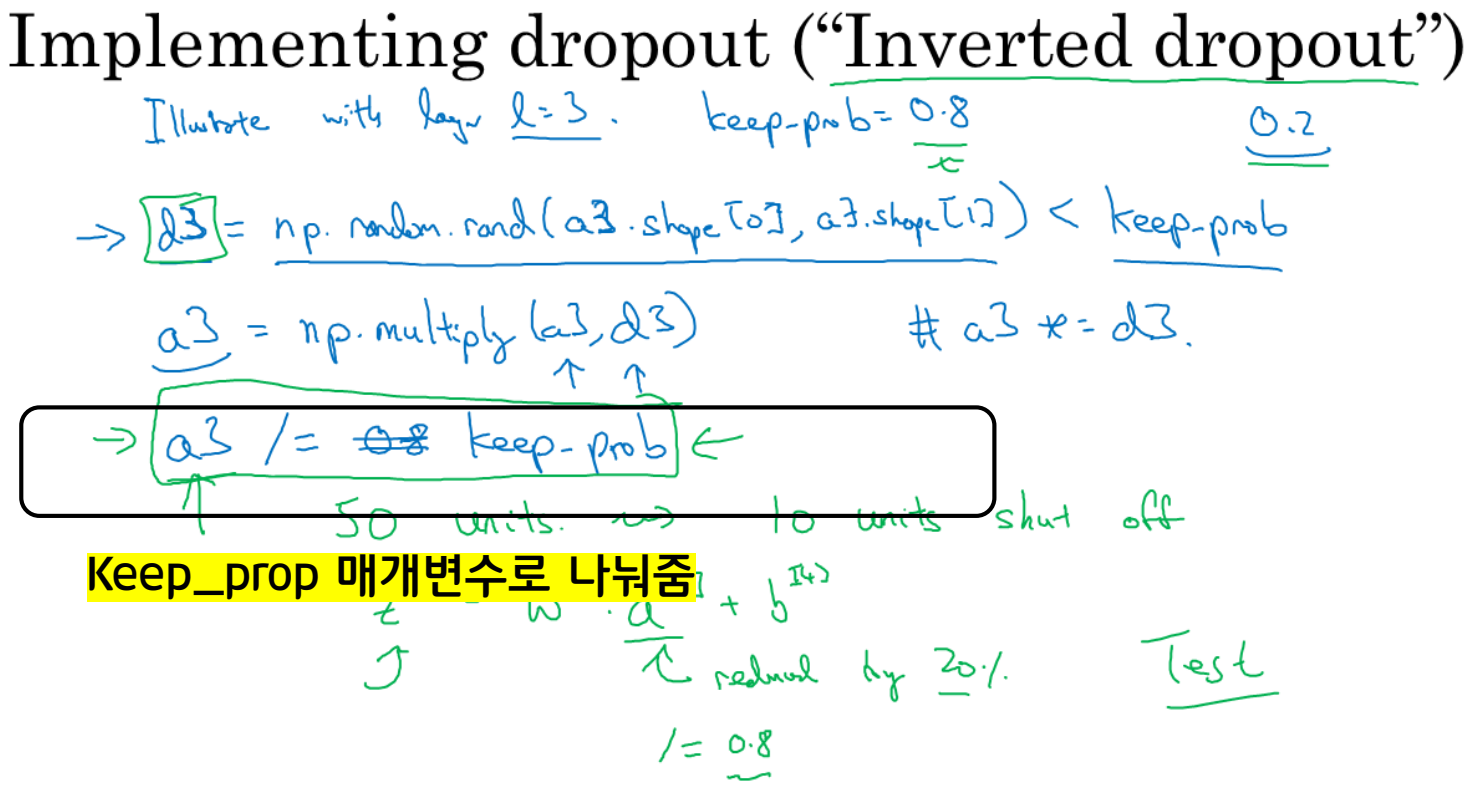


Andrew Ng

Implementing dropout("Inverted dropout") 역드롭아웃

일반적인 기법 역 드롭아웃

층이 3인 경우를 예시로 단일 층에서 드롭아웃을 구현하는 방법

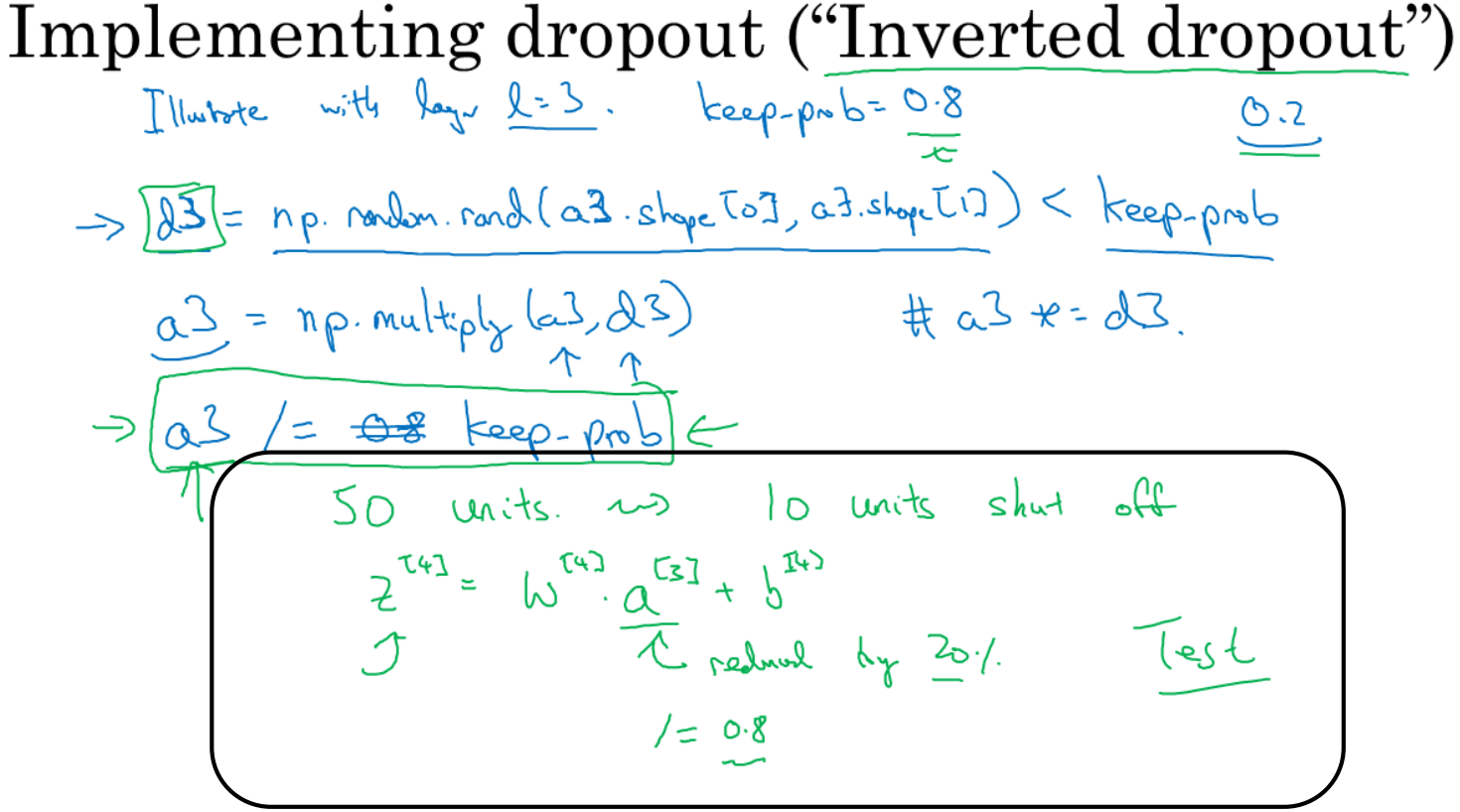


Andrew Ng

Implementing dropout("Inverted dropout") 역드롭아웃

일반적인 기법 역 드롭아웃

층이 3인 경우를 예시로 단일 층에서 드롭아웃을 구현하는 방법



Andrew Ng

세번째 은닉층에 50개의 유닛, 즉 50개의 뉴런이 있음
a3은 (50,1)차원을 벡터화해서 평균적으로 10개 유닛 삭제
z4의 기대값을 줄이지 않기 위해 이 값을 0.8로 나눠줘야
그래야 필요한 20%의 값을 원래대로 만듬

Implementing dropout("Inverted dropout")
역드롭아웃

반복마다 0이 되는 은닉 유닛은 무작위로 달라져야 함

경사하강법의 하나의 반복마다 0이 되는 은닉 유닛들이 달라짐

세번째 층의 d3벡터는 어떤 노드를 0으로 만들지 결정

정방향 역전파 모두

Implementing dropout("Inverted dropout") 역드롭아웃

Making predictions at test time

테스트에서는 드롭아웃을 사용하지 않음

테스트는 예측하는 것이므로 결과가 무작위로 나오는 것을 원하지 않음

테스트에서 드롭아웃을 구현하지 않아도 활성화 기대값의 크기는 변하지 않기 때문에 스케일링 필요 없음

Understanding dropout

왜 드롭아웃이 잘 돌아갈까

Intuition : can’t rely on any one feature, so have to spread out weights

더 작은 신경망을 사용하는 것이 정규화의 효과를 주는 것처럼 보인다

Understanding dropout

왜 드롭아웃이 잘 돌아갈까

드롭아웃을 통해 무작위로 삭제되기 때문에 어떤 특성에도 의존할 수 없다

그 트성이 무작위로 바뀌거나 특성의 고유한 입력이 무작위로 바뀔 수 있음

특정 입력에 모든 것을 걸 수 없음

네 개의 입력에 각각의 가중치를 분산시킴

Understanding dropout

가중치를 분산시킴으로써 가중치의 노름의 제곱값이 줄어들게 됨 – shrink weights

L2정규화에서 봤던 것처럼 드롭아웃을 구현하는 효과는 가중치를 줄이는 것이고
L2 정규화처럼 과대적합을 막는 데 도움이 됩니다

L2정규화에서 다른 가중치는 다르게 취급, 그 가중치를 곱해지는 활성화의 크기에 따라 다름

드롭아웃은 L2정규화와 비슷한 효과를 보여줄 수 있음
L2정규화가 다른 가중치에 적용된다는 것과 서로 다른 크기의 입력에 더 잘 적응함

Understanding dropout

매개변수 keep_prob 층마다 바꾸는 것 가능

W2 shape이 (7,7)로 가장 많은 매개변수를 갖기 때문에 이 행렬의 과대적합을 줄이기 위해
이 층은 상대적으로 낮은 keep_prob을 가져야 함

과대적합의 우려가 적은 층에서는 더 높은 keep_prob을 설정해도 됨
과대적합의 우려가 없는 층은 keep_prob을 1로 설정해도 됨

매개변수가 많은 층, 즉 과대적합의 우려가 많은 층은 더 강력한 형태의 드롭아웃을 위해
Keep_prob을 작게 설정

Understanding dropout

입력층에 대해서는 keep_prob 1.0이 가장 흔한 값

0.9도 사용... 1에 가까운 값 사용

컴퓨터 비전에서 데이터가 적어 거의 대부분 과대적합이 일어나고 그래서 드롭아웃을 많이 사용한다

Understanding dropout

Dropout의 큰 단점은 비용함수 J가 더 이상 잘 정의되지 않음

모든 반복마다 무작위로 한 뭉치의 노드들을 삭제하게 됨

따라서 경사 하강법의 성능을 다중으로 확인한다면,

모든 반복에서 잘 정의된 비용함수 J가 하강하는지 확인하는게 어려워짐

그래서 최적화되는 비용함수는 잘 정의되지 않아 계산하기 어려움

어떤 모양인지 확인해서 디버깅 하는 것도 어려움

보통 keep_prob을 1로 설정해서 드롭아웃 효과를 멈추고 코드를 실행시켜 J가 단조감소하는지 확인

그리고 드롭아웃 효과를 다시 주고 드롭아웃이 있을 때 코드를 바꾸지 않도록 함

Other Regularization methods

고양이 분류기를 훈련시키는 경우 더 많은 훈련 데이터가 과대적합을 해결하는 데 도움을 줄 수 있지만
많은 비용이 들어가거나 불가능한 경우가 있음
수평방향으로 뒤집은 이미지를 훈련세트에 추가시켜 훈련 세트를 늘리는 방법도 잇음
새로운 m개의 독립적인 샘플을 얻는 것보다 이 방법은 중복된 샘플들이 많아져서 좋지 않다
그러나 새로운 고양이 사진을 더 많이 구하지 않고 할 수 있는 방법

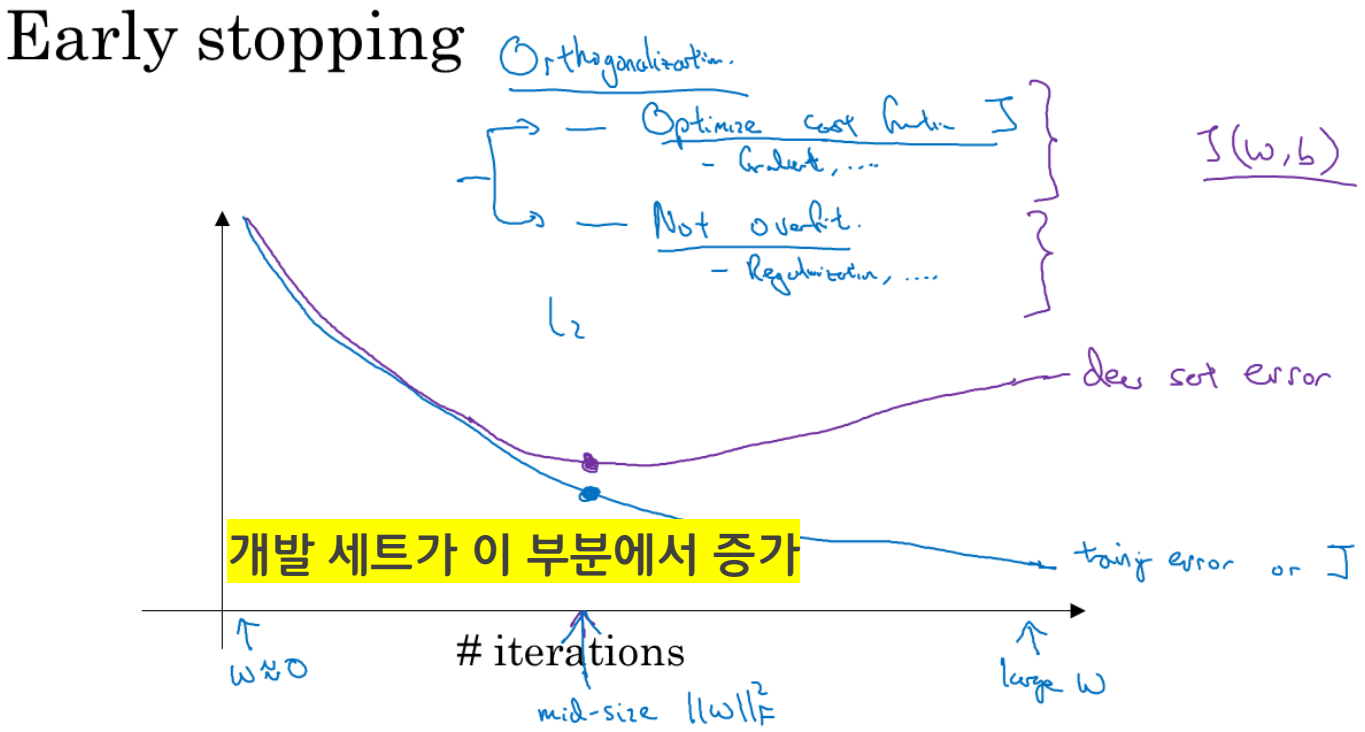
이미지의 무작위적인 왜곡과 변형을 통해 데이터 세트를 증가시키고 추가적인 훈련 셈플을 얻을 수 있다
완전히 새로운 독립적인 고양이 샘플을 얻는 것보다 더 많은 정보를 추가해주지 않을 것이다
그러나 컴퓨터적인 비용이 들지 않고 할 수 있다

Other Regularization methods

합성한 이미지를 통해 사용하는 알고리즘에게 고양이 이미지를 뒤집어도 여전히 고양이라는 것을 학습시킬 수 있다 뒤집어진 고양이는 원하지 않으므로 수직방향으로는 뒤집지 않는다

데이터 증가는 정규화 기법과 비슷하게 사용할 수 있다

Early stopping



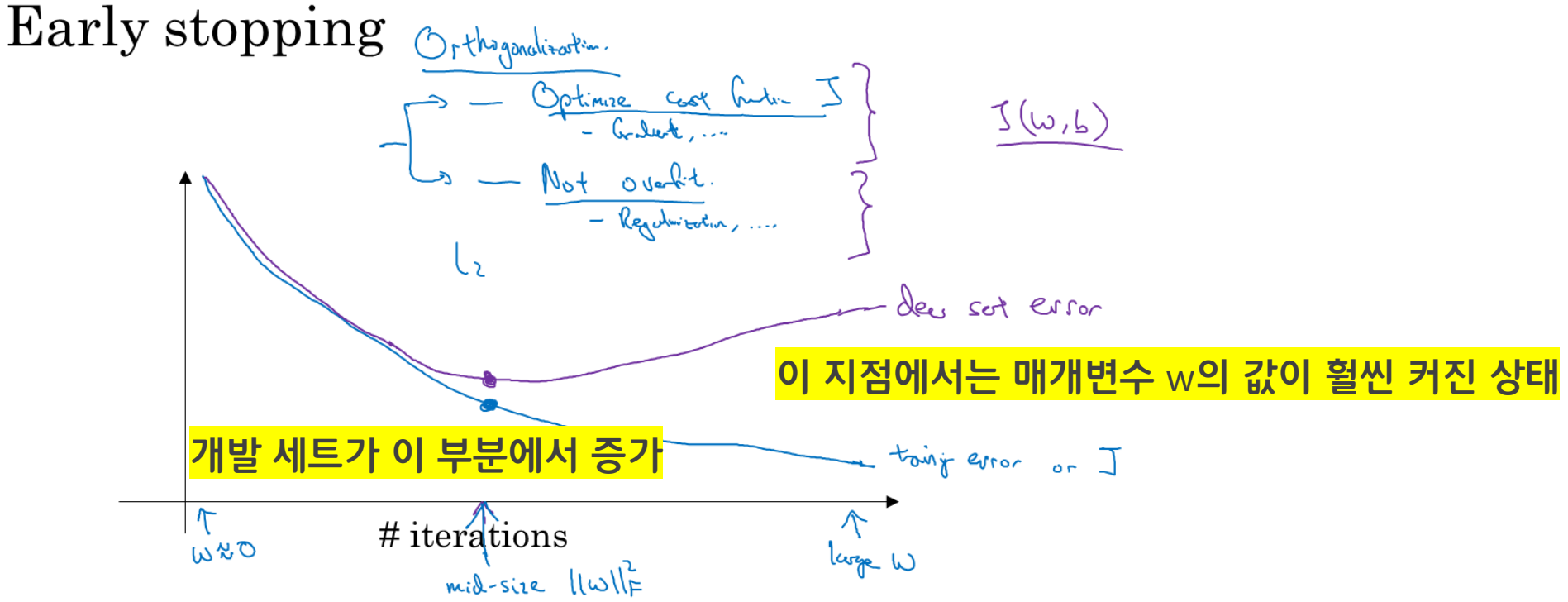
Andrew Ng

경사하강법을 실행시켜 훈련세트에 대한 분류 오차, 즉 훈련 오차를 그리거나 최적화하는 비용함수 J를 그리게 된다.

신경망의 많은 반복을 시키지 않은 경우 매개변수화는 0에 가깝다 무작위의 작은 값으로 초기화시켜서

오랜시간 훈련시키기 전까지 w는 여전이 작다

Early stopping



이 지점에서는 매개변수 w의 값이 훨씬 커진 상태

개발 세트가 이 부분에서 증가

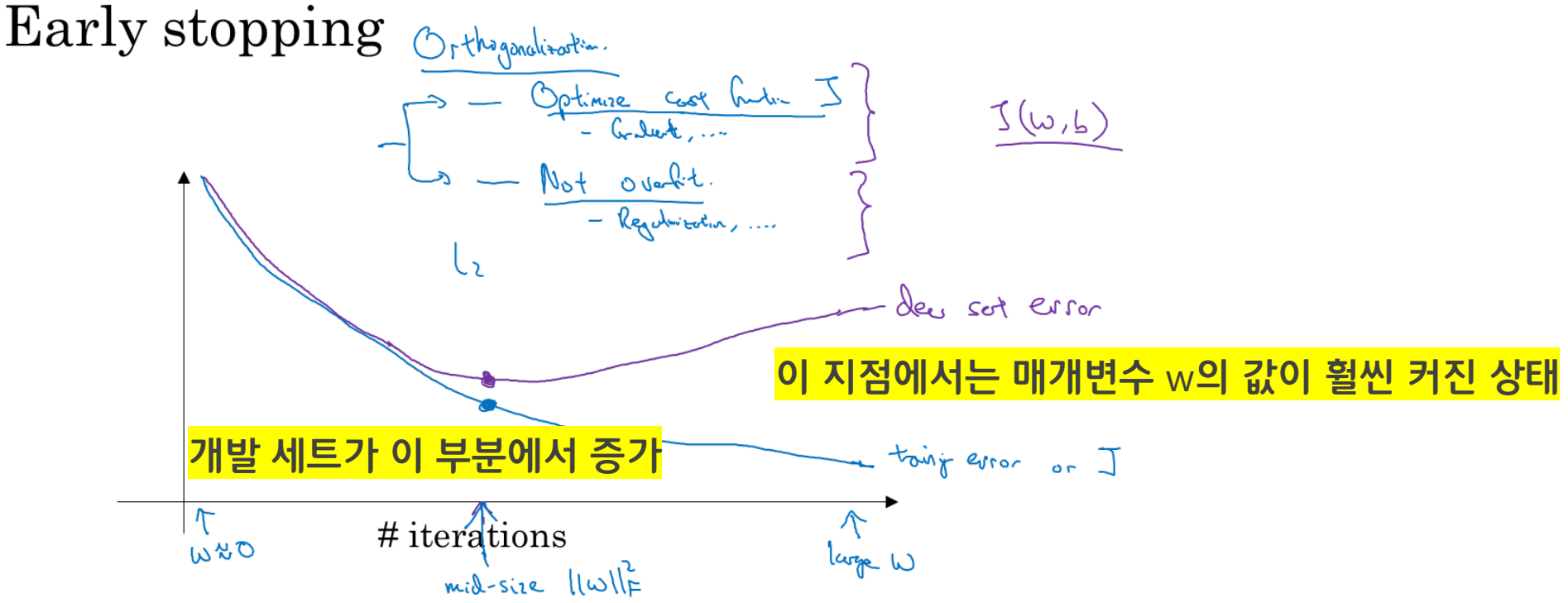
Andrew Ng

훈련오차나 비용함수 J는 단조감소하는 형태로 그려져야함

조기 종료에서는 개발 세트 오차를 함께 그려줌

반복을 실행할 수록 w의 값은 계속 커짐

Early stopping



이 지점에서는 매개변수 w의 값이 훨씬 커진 상태

개발 세트가 이 부분에서 증가

Andrew Ng

조기종료 기법에서 반복을 중간에 멈추면, w가 중간 크기의 값을 갖는 형태

L2정규화와 비슷하게 매개변수 w에 대해 더 작은 노름을 갖는 신경망을 선택함으로써 신경망이 덜 과대적합

Early stopping

1) 비용함수 J를 최적화하는 알고리즘을 원한다

- 경사하강법처럼 이를 위한 몇가지 알고리즘이 있다 (모멘텀 , RMSProp,Adam)

비용함수J를 최적화하고 난 뒤에 과대적합되는 것을 막기 위한 몇가지 도구들이 또 있다(정규화, 데이터 추가하기)

따라서 비용함수 J를 최적화하는 하나의 도구세트만 있다면 머신러닝이 훨신 더 간단해짐

비용함수 J를 최적화할 때 집중하는 것은 J(w,b)가 가능한 작아지는 값을 차즌ㄴ것 외에 신경쓰지 않는다

과대적합을 막는 것은 다른일! = 분산을 줄임!

조기종료의 주된 단점을 이 둘을 섞어버림 → 문제를 독립적으로 다룰 수 없음

Early stopping

왜냐하면 경사 하강법을 일찍 멈춤으로써 비용함수 J를 최적화하는 것을 멈추게 됨
비용함수 J를 줄이는 일은 잘하지 못함, 동시에 과대적합을 막으려고 함

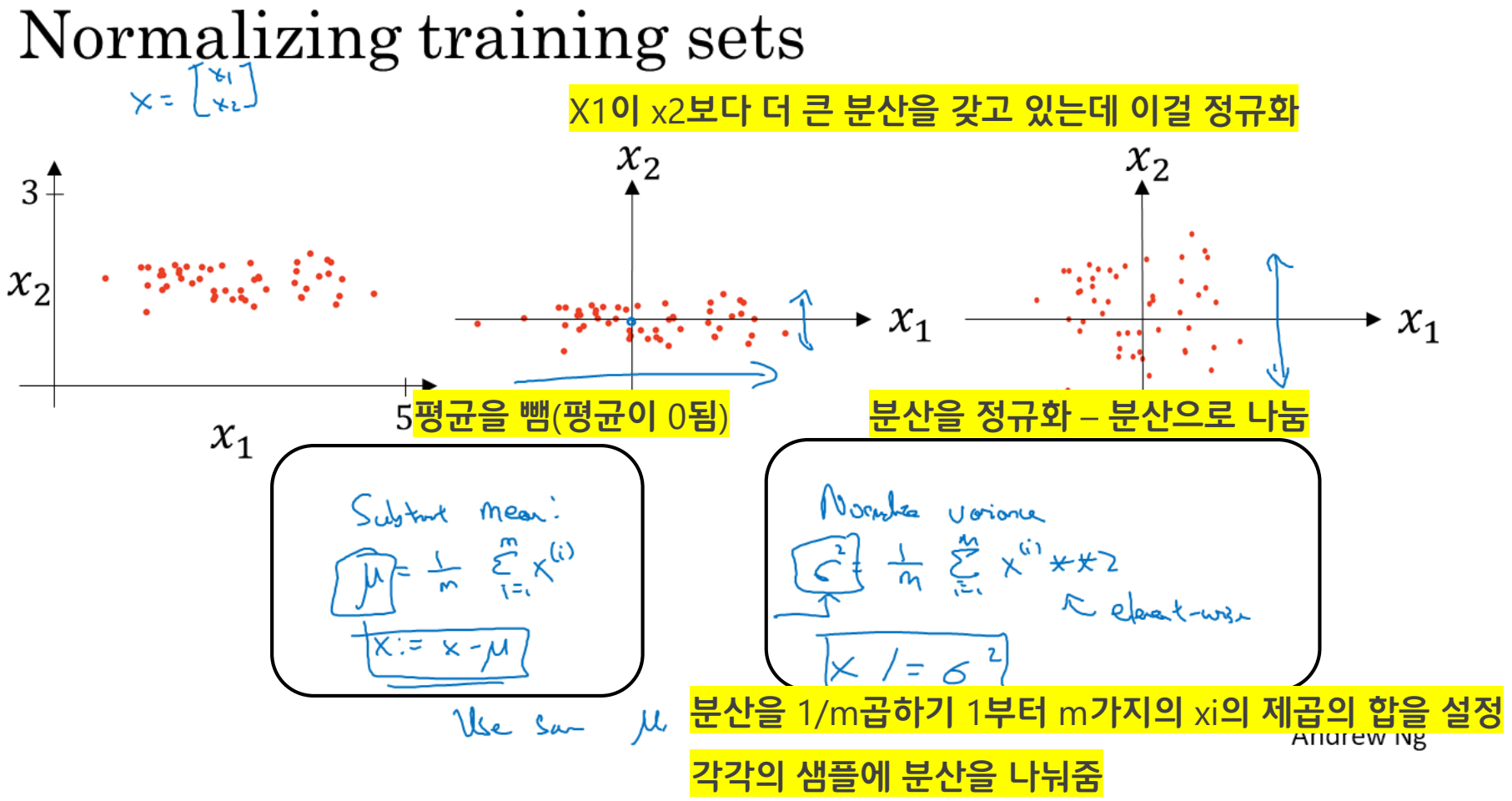
따라서 서로 다른 도구를 사용하는 대신 혼합된 도구를 사용하게 됨
문제를 더 복잡하게 만듬, 조기종료를 사용하는 것의 대안은 L2 정규화를 사용하는 것

그럼 가능한 오래 신경망을 훈련시킬 수 있게 됨
이를 통해 하이퍼파라미터의 탐색 공간이 더 분해하기 쉽고 찾기 쉬워짐
단점은 정규화 매개변수 람다에 많은 값을 시도해야 함
람다의 많은 값을 대입하는 것은 컴퓨터적으로 비용이 많이 듬

조기종료의 진짜 장점은 경사하강법 과정을 한번만 실행해서 작은 w, 중간 w, 큰 w의 값을 얻게 됨
많은 시도를 할 필요 없음, 훈련을 빠르게 시키기 위한 최적화

Normalizing Inputs

신경망의 훈련을 빠르게 할 수 있는 하나의 기법은 정규화하는 것



Normalizing Inputs

이것을 훈련 데이터를 확대하는 데 사용한다면, 테스트 세트를 정규화 할 때도
같은 평균과 분산을 사용하라, 훈련세트와 테스트세트를 다르게 정규화하고 싶지는 않을 것이다
평균과 분산의 값이 어떤 값이든 이 두가지 식에 사용

훈련 세트와 테스트 세트에서 별개로 평균과 분산을 추정하는 대신에

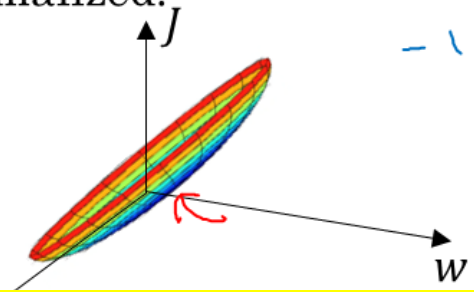
훈련샘플과 테스트샘플 모두 훈련 데이터에서 계산된 것과 같은 평균과 분산에 의해 정의된 변형을
거치기를 원하기 때문

Why normalize inputs?

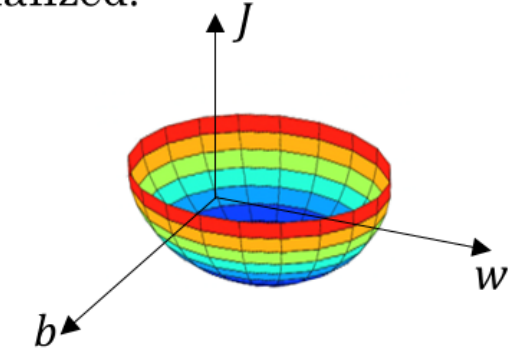
Why normalize inputs?

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Unnormalized:

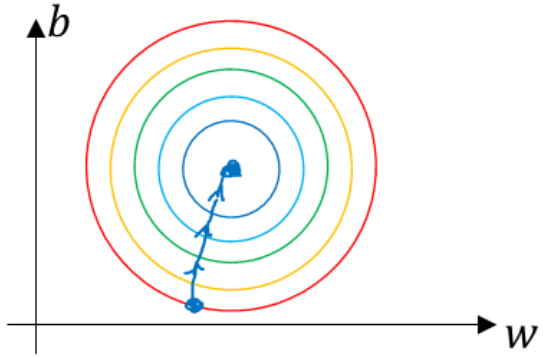


Normalized:



정규화되지 않은 입력특성을 사용하면 비용함수가 이렇게 됨

매우 구부러진 활처럼 가늘고 긴 모양의 비용함수



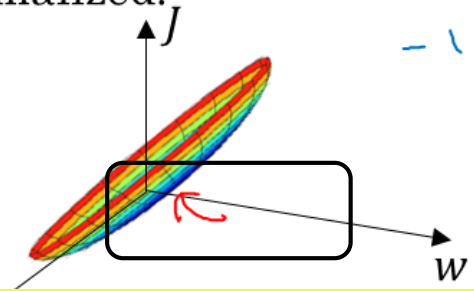
Andrew Ng

Why normalize inputs?

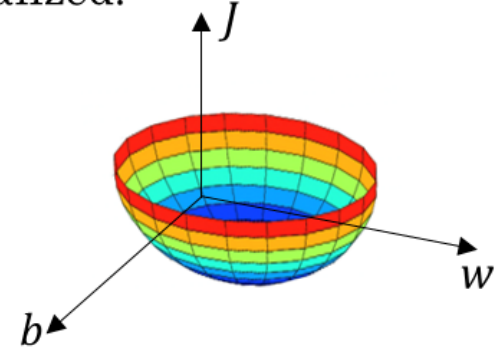
Why normalize inputs?

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Unnormalized:



Normalized:

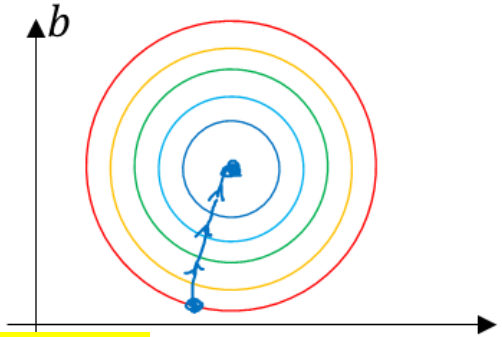


여러분들이 찾으려는 최소값은 이 부분에 위치

만약 특성들이 매우 다른 크기를 갖고 있다면...

X1 = 1....1000

X2 = 0....1



매개변수에 대한 비율 값의 범위는 w1과 w2가 굉장히 다른 값을 갖게 됨

이 축이 w1과 w2가 될 것, 비용함수는 이처럼 매우 가늘고 긴 모양

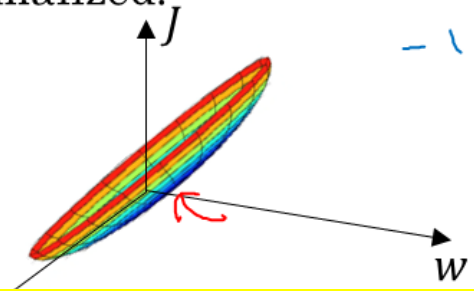
Andrew Ng

Why normalize inputs?

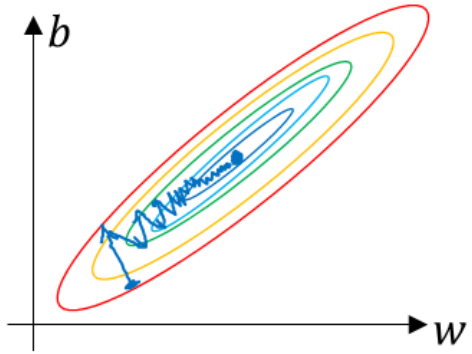
Why normalize inputs?

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

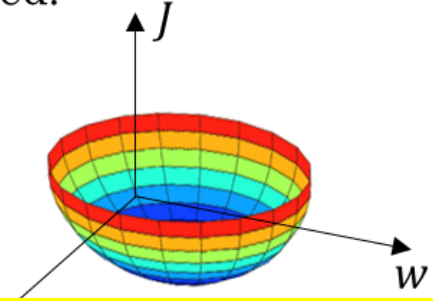
Unnormalized:



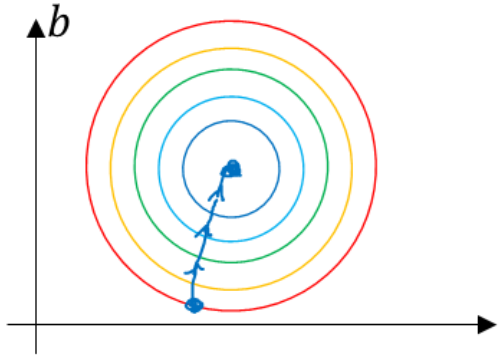
이 함수의 등고선을 그려보면 가늘고 긴 함수를 얻음



Normalized:



특성을 정규화하면 비용함수는 평균적으로 대칭적 모양



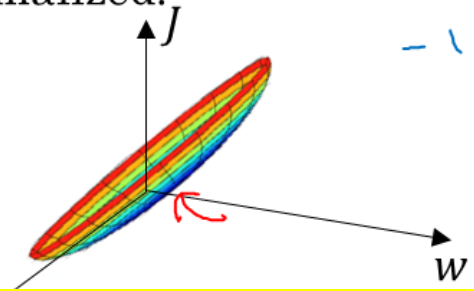
Andrew Ng

Why normalize inputs?

Why normalize inputs?

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

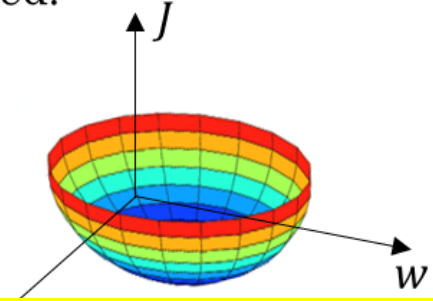
Unnormalized:



이 함수의 등고선을 그려보면 가늘고 긴 함수를 얻음

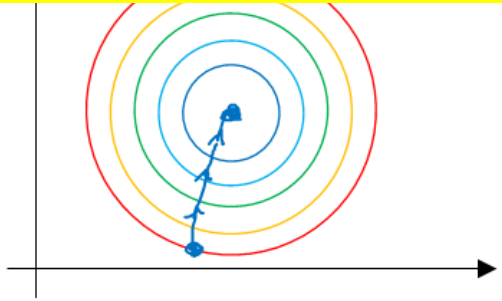
경사하강법 실행 시 최소값을 갖기 위해 많은 단계가 필요

Normalized:



특성을 정규화하면 비용함수는 평균적으로 대칭적 모양

어디서 시작하든 경사하강법은 최소값으로 바로 갈 수 있음



Andrew Ng

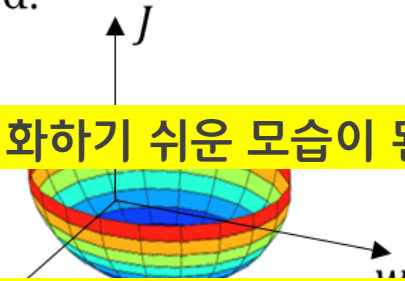
Why normalize inputs?

Why normalize inputs?

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

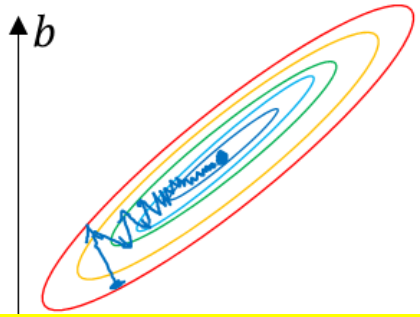
Unnormalized:

Normalized:



특성이 비슷한 크기를 갖을 때 비용함수가 더 둥글고 최적화하기 쉬운 모습이 된다는 대략적인 직관을 얻음

대부분 -1에서 1혹은 서로 비슷한 분산으로 비용함수 J를 최적화하기 쉽고 빠르게 만듬



X1 : 0..1

X2 : -1...1

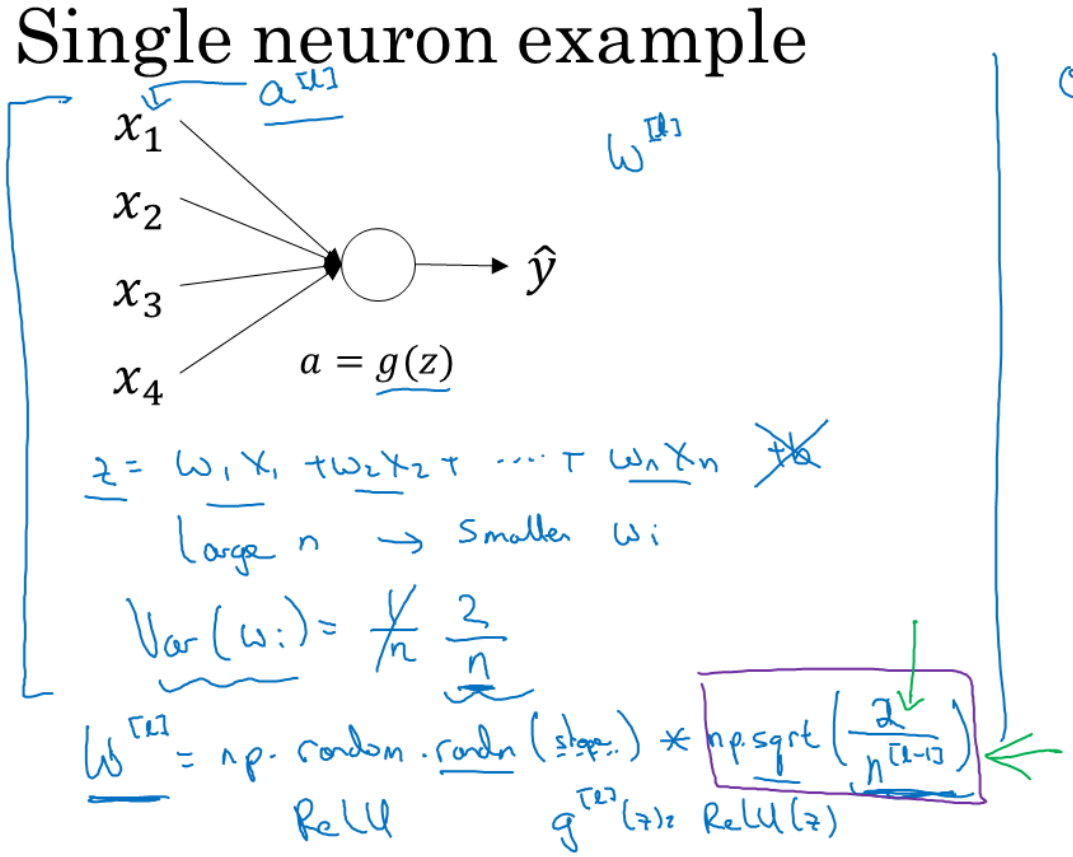
X3: 1...2

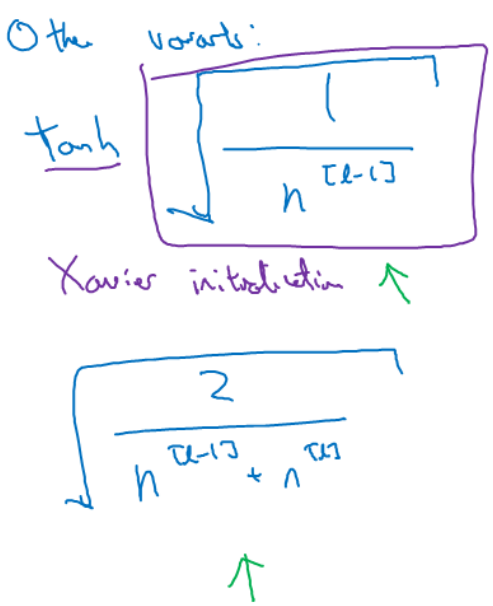
이들은 상당히 비슷한 범위

모든 _특성을 비슷한 크기로 보장할 수 있는 분산을 설정하면 학습 알고리즘이 빠르게 실행 됨

무조건 하셈 손해는 없음!

Weight initialization for deep networks



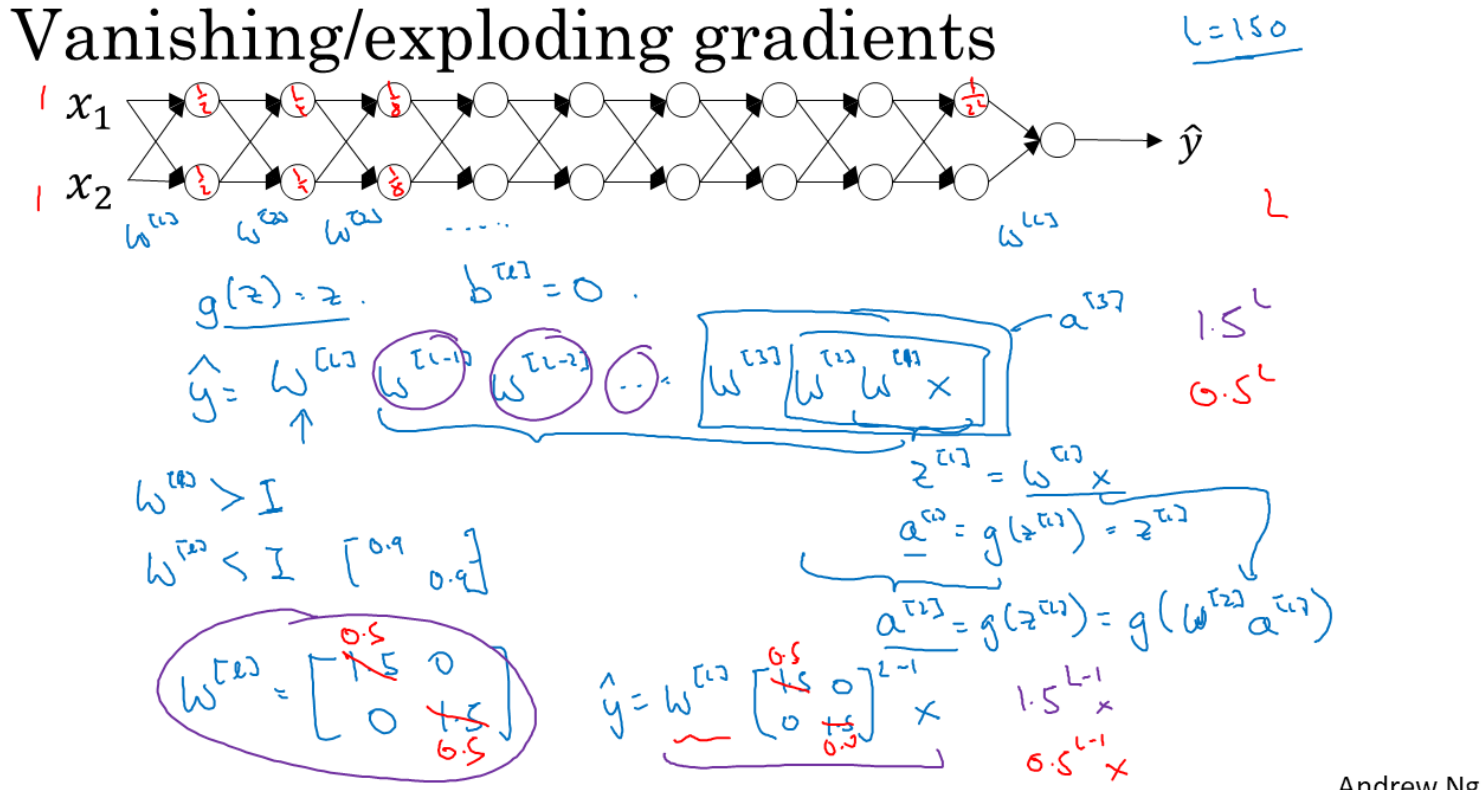


Andrew Ng

activation function**에 맞게** weight **초기화를 해준다. 표준정규분포에서 뽑은 숫자에 특정 수를 곱해주어 분산을 조정한다. - 초기화를 잘해주면** Gradients Vanishing/exploding**에 도움이 된다. - 하지만 다른** parameter **조정에 비해 덜 중요하다.**

Vanishing/explding gradients

T신경망을 훈련시키는 것, 특히 매우 깊은 신경망을 훈련시키는 것의 문제는 경사의 소실과 폭발
매우 깊은 신경망을 훈련시킬 때 미분값 혹은 기울기가 아주 작아지거나 커질 수 있음



Andrew Ng

깊은 신경망에서 L의 값이 크다면 y의 예측값도 매우 커질 것이다 – 폭발
깊은 신경망에서 L의 값이 작다면 y의 예측값도 매우 작을 것이다 - 소실

Vanishing/explding gradients

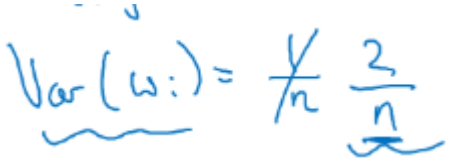
W^l이 단위행렬보다 조금 더 크다면 매우 깊은 네트워크의 경우 활성화값 폭발 가능
W^l이 단위행렬보다 조금 작다면 활성화값 기하급수적으로 감소

미분값 즉, 경사하강법에서 계산하는 경사가 층의 개수에 대한 함수로
기하급수적으로 증가하거나 감소한다는 것을 보여주는데 사용할 수도 있음

부분적인 해결책 : 가중치를 어떻게 초기화시키나

Vanishing/explding gradients

ReLU 활성화 함수를 사용하는 경우
분산이 1/n보다 2/n으로 설정하는 것이 더 잘 작동함



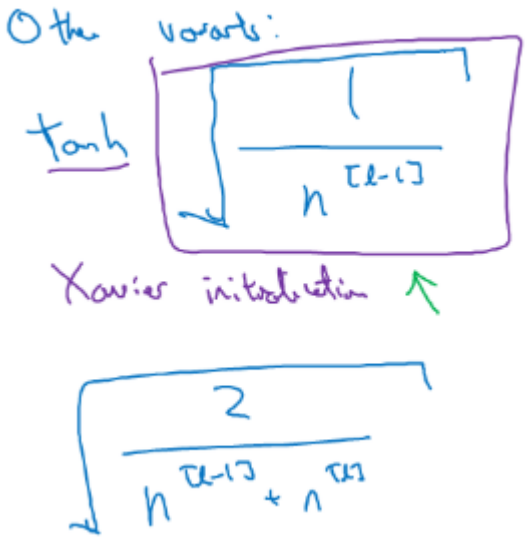
n^l-1을 사용하는 이유는 더 일반적인 경우에 층 l은 해당 층의 각 유닛에 대해 n^l-1의 입력을 갖음
따라서 입력 특성 혹은 활성화값의 평균이 대략 0이고 표준편차가 1을 갖는 다면 비슷한 크기를 갖음

경사 소실과 폭발 문제에 확실히 도움을 줄 수 있음
왜냐하면 각각의 가중치 행렬 w를 1보다 너무 커지거나 너무 작아지지 않게 설정해서 너무 빨리 폭발하거나 소실되지 않게 함

Vanishing/explding gradients

다른 변형

그 전까지는 ReLu 활성화 함수를 사용하는 상황을 가정
Tanh 활성화 함수를 사용한다면, 상수 2대신 상수 1을 사용하자는 논문



세이비어 초기화
Xavier initalization

실제로 이 모든 식들은 그저 시작점을 제공
가중치 행렬의 초기화 분산에 대한 기본값을 줄 뿐

분산 매개변수는 하이퍼파라미터로 조정할 또 다른 값이 됨

Vanishing/explding gradients

이 조정도 효과가 있을 수 있으나 조정을 시도하는 첫번째 하이퍼파라미터는 아님

그러나 그 조정이 상당히 도움이 되는 경우가 많이 있음

Gradient-Checking

https://goodtogreate.tistory.com/entry/Gradient-Checking

Mini-batch gradient descent

벡터화가 m개의 샘플에 대한 계산을 효율적으로 만들어줌

그러나 m이 크다면 여전히 느림

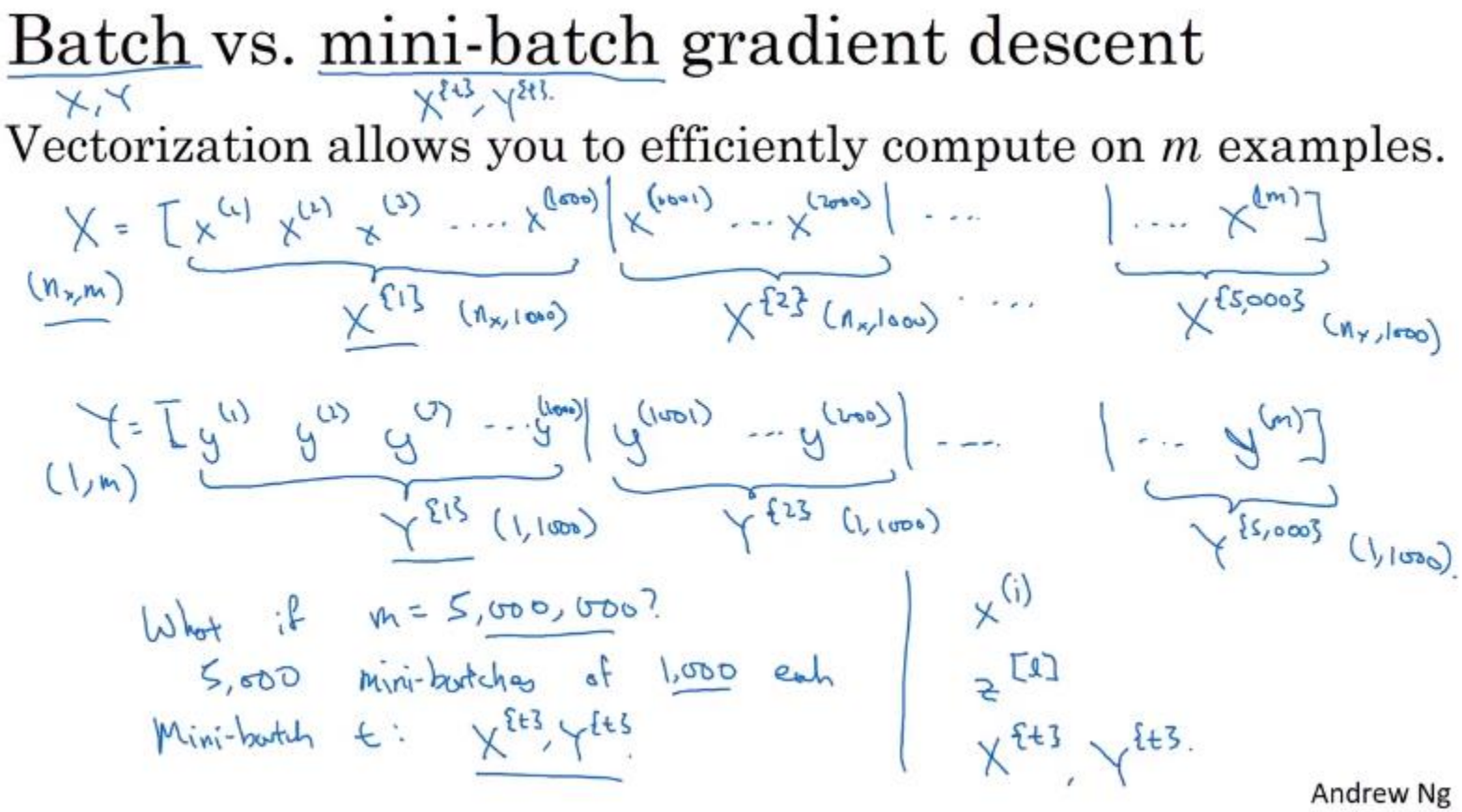
전체 훈련 세트에 대한 경사 하강법을 구현하면 경사 하강법의 한 단계를 밟기 전에 모든 훈련 세트를 처리해야 함

또 경사하강법의 다음다음 단계를 밟기 전에 다시 오백만 개의 전체 훈련 샘플을 처리해야 함

따라서 오백만개의 거대한 훈련 샘플을 모두 처리하게 전에 경사 하강법이 진행되도록 하면 더 빠른 알고리즘을 얻을 수 있음

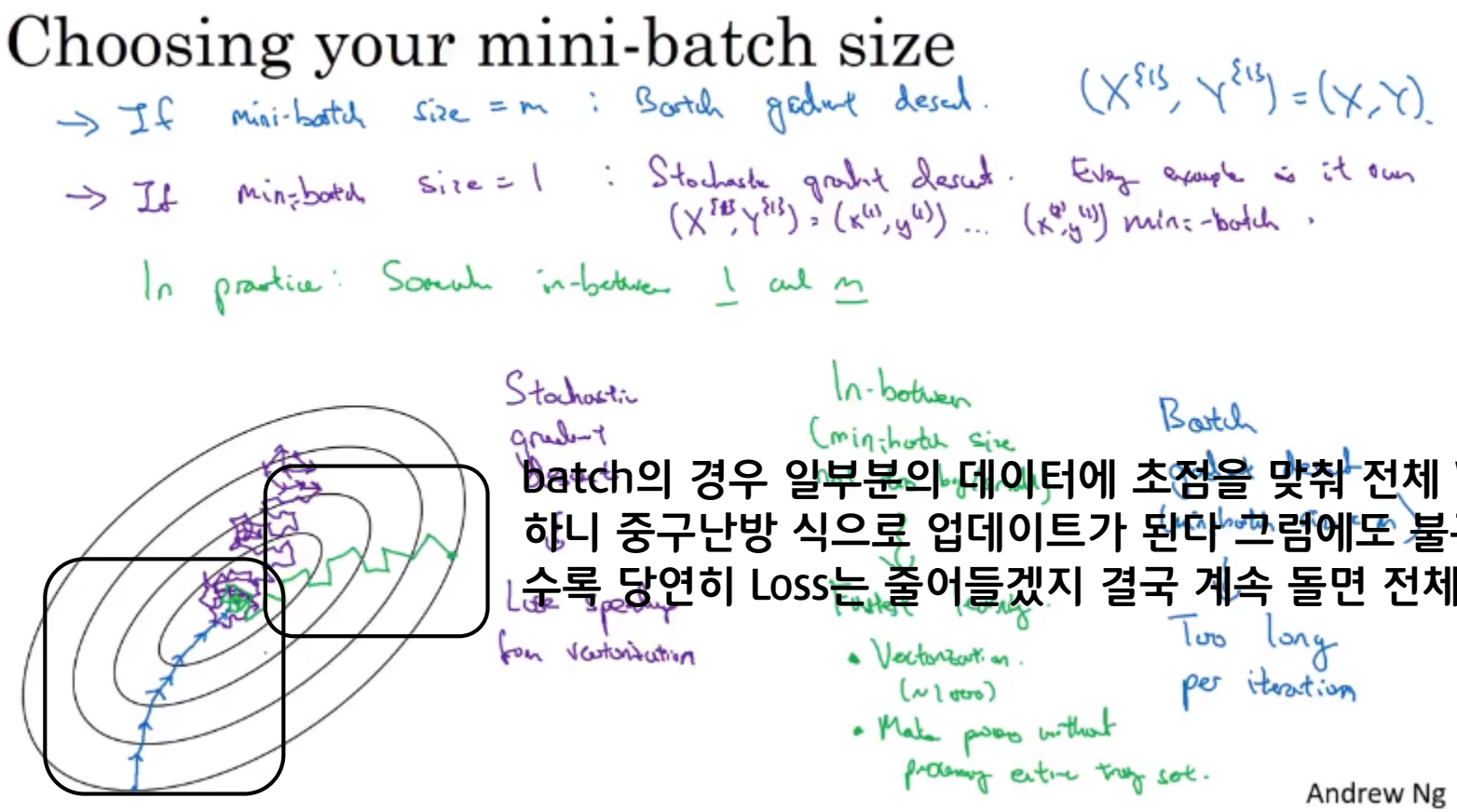
훈련 세트를 5000개 씩 나눠서 1000번 진행

Mini-batch gradient descent



Andrew Ng

Understanding mini-batch gradient decent



batch의 경우 일부분의 데이터에 초점을 맞춰 전체 W값을 업데이트 하니 중구난방 식으로 업데이트가 된다 그럼에도 불구하고 시간이 갈수록 당연히 Loss는 줄어들겠지 결국 계속 돌면 전체가 되니

기존 batch (**전체데이터 기준**) gradient decent의 경우 loss값이 전체 트레이닝셋(평균)에 대한 W값의 업데이트 이므로 항상 낮아지는 반면

Understanding mini-batch gradient decent

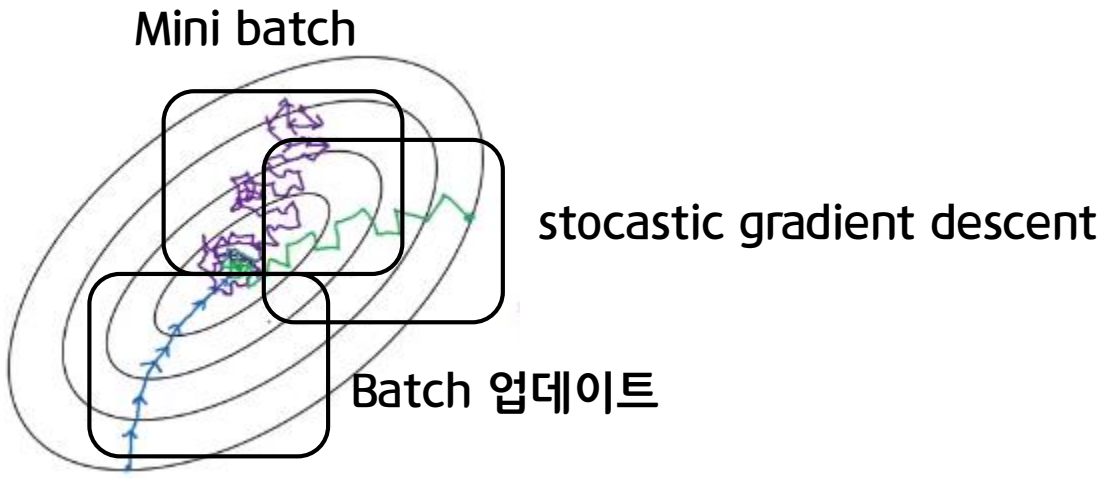
배치 경사하강법에서는 모든 반복에서 전체 훈련 세트를 진행하고 비용함수 그래프를 RFUT을 때 모든 반복마다 감소해야 함 하나라도 올라간다면 무언가 잘못된 것

미니배치 경사하강법에서 비용함수에 대한 진행을 그려본다면 모든 반복마다 감소하지 않음
모든 반복에서 다른 훈련세트, 즉 다른 미니배치에서 훈련하기 때문.
비용함수 J : 전체적인 흐름은 감소하나 약간의 노이즈 발생.
따라서 미니배치 경사 하강법을 훈련하기 위한 J^t를 그리면 여러 에포크를 거쳐서 진동함
모든 반복에서 아래로 내려가지 않음
약간의 노이즈기 발생하는 이유는 X^1과 Y^1이 상대적으로 쉬운 미니배치라서 비용이 약간 낮은데,
우연적으로 X^2와 Y^2가 더 어려운 미니배치라서 아마 잘못 표시된 샘플이 있다든지의 이유로
비용이 약간 높아질 수 있음

Understanding mini-batch gradient decent

선택해야 하는 매개변수 : 미니배치 크기

여기서 위에서 정의한 mini-batch 사이즈 t = 1로 하면 각 training예제 마다 W를 업데이트 한다는건데, 이런걸 stocastic gradient descent라고 한다. 혹은 online learning (즉시 학습) 이라고 하는것 같은데 정확히 두 용어의 차이점은 모르겠으나 전자는 W업데이트 방법만에 초점을 두어 말하는 것같고 후자는 전체 기계학습의 방법을 의미하는듯하다.



Understanding mini-batch gradient decent

배치 경사 하강법은 어딘가에서 시작해 상대적으로 노이즈가 작고 큰 단계를 취함

최소값으로 나아감

그와 반대로 확률적 경사 하강법에서는 어딘가에서 시작하면, 모든 반복에서 하나의 훈련 샘플로 경사 하강법을 실행

대부분 전역 최솟값으로 가게 되지만 어떤 경우는 이처럼 잘못된 방향을 가르켜 잘못된 곳으로 가기도 함

따라서 확률적 경사 하강법은 극단적으로 노이즈가 많을 수 있지만, 평균적으로는 좋은 방향으로 가게됨

잘못된 방향일 수도 있음!

따라서 확률적 경사 하강법은 진동하면서 최솟값 주변을 돌아다니게 되지만, 최솟값으로 곧장 가서 머물지는 않을 것

Understanding mini-batch gradient decent

미니배치 크기는 1(작은 값)과 m(큰값) 사이

Batch gradient 배치 경사하강법을 사용한다면, 미니배치 크기는 m과 같음
그럼 매우 큰 훈련 세트를 모든 반복에서 진행
이것은 주된 단점 → 너무 오랜 시간 걸림

확률적 경사 하강법을 사용한다면, 하나의 샘플만을 처리한 뒤에 계속 진행할 수 있음
노이즈로 작은 학습률을 통해 줄일 수 있음
그러나 확률적 경사하강법의 큰 단점은 벡터화에서 얻을 수 있는 속도 향상을 잃게 됨
한번에 하나의 훈련세트를 진행하기 때문에 각 샘플을 진행하는 방식이 매우 비효율적

따라서 제일 잘 작동하는 것은 1~m사이에 있는 값

Understanding mini-batch gradient decent

그리고 실제로 이것은 가장 빠른 학습을 제공

하나는 많은 벡터화를 얻는다는 것

미니배치 크기가 1000개의 샘플이라면, 1000개의 샘플에 벡터화를 하게 될 것임

한 번에 샘플을 진행하는 속도가 더 빨라지게 됨

두번째로 전체 훈련 세트가 진행되기를 기다리지 않고 진행 가능

각각의 훈련세트의 에포크는 5000번 경사 하강

Understanding mini-batch gradient decent

항상 최솟값으로 수렴한다고 보장 할 수 없지만 더 일관되게 전력의 최솟값으로 향하는 경향이 있음

그리고 매우 작은 영역에서 항상 정확하게 수렴하거나 진동하게됨
미니배치그 크기가 m이나 1dl 아닌 그 사이의 값이어야 한다면 이 값은 어떻게 정함

첫번째로 작은 훈련 세트라면 그냥 배치 경사 하강법 사용
훈련 데이터가 작다면, 미니배치 경사 하강법을 사용할 필요 없이 전체 훈련 세트를 빠르게 진행 가능
작은 훈련 세트 (샘플이 2000개보다 작은 경우)

이와 달리 더 큰 훈련세트라면 전형적인 미니배치 크기는 64에서 512
컴퓨터 메모리의 접근 방식을 생각해보면, 미니배치 크기가 2의 제곱인 것이 코드를 빠르게 실행시켜줌

Understanding mini-batch gradient decent

미니배치에서 X^t, Y^t가 CPU와 GPU 메모리에 맞는지 확인하세요

훈련 샘플 크기에 달려있음

CPU와 GPU 메모리에 맞지 않는 미니배치를 진행시키면, 성능이 갑자기 떨어지고 훨씬 나빠짐

하이퍼파라미터 미니배치 크기는 몇가지 다른 2의 제곱수를 시도해보고

경사 하강법 최적화 알고리즘을 가능한 가장 효율적으로 만드는 값을 선택

Exponentially weighted averages (지수 가중 이동 평균)

학습을 조금 더 빠르게 할수 있는 방법이 없을까? 해서 고안된 방법이 momentum (바로아래나옴) 이라는 방법인데 여기에 사용된 방법(?) 알고리즘 이다. 앞서 설명한 mini-batch gradient를 사용하면 각 gradient descent가 울퉁불퉁 한 모양을 띄게 되었는데 이게 사실 위 사진에 **등고선을 넘어가 학습이 되지 않는 문제**를 이르킬 수 도 있고 **왓다리 갔다리 하면서 학습 속도가 느려질 수 있다**. 이걸 조금 더 스무스 하게 batch gradient가 한것처럼 하려면 어떻게 해야할까? 전 단계의 상태를 참조해서 전단계와의 갭(이동거리 및 방향)을 줄이면서 학습하면 되지 않을까?

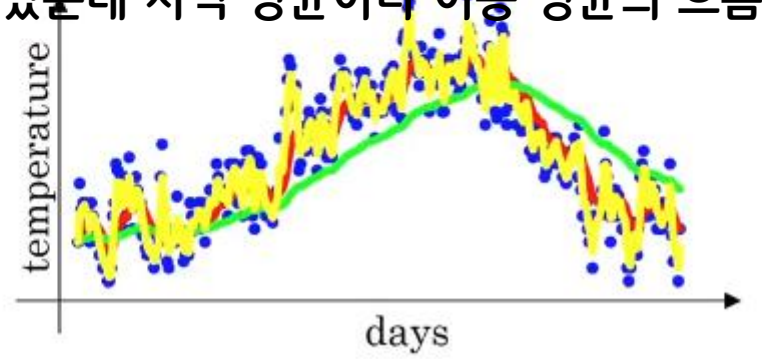
Exponentially weighted averages (지수 가중 이동 평균)

Exponentially weighted averages

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

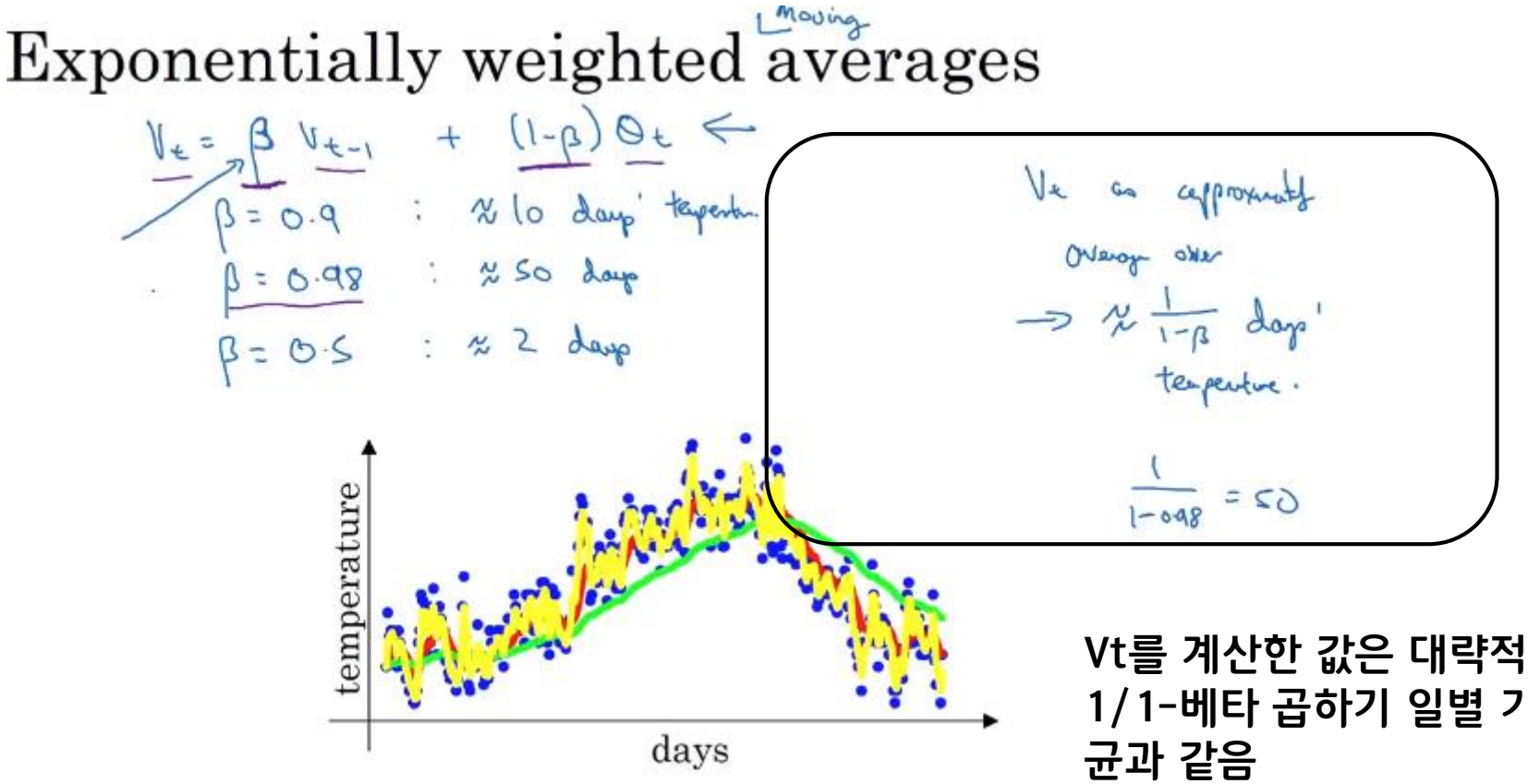
β (보통 0.9를 사용) 가 나오는데 이값에 전단계의 계산값을 곱하고 (1-β)에 이번 계산값 곱해서 **이전 값을 통해 현재 값이 너무 튀는걸 어느정도 보정**하고 **현재값도 어느정도 움직임에 영향을 미칠 수** 있도록 하는 방법

데이터에 약간의 노이즈가 있는데 지역 평균이나 이동 평균의 흐름을 계산하고 싶다면 지수가중이동평균을 계산한다



Andrew Ng

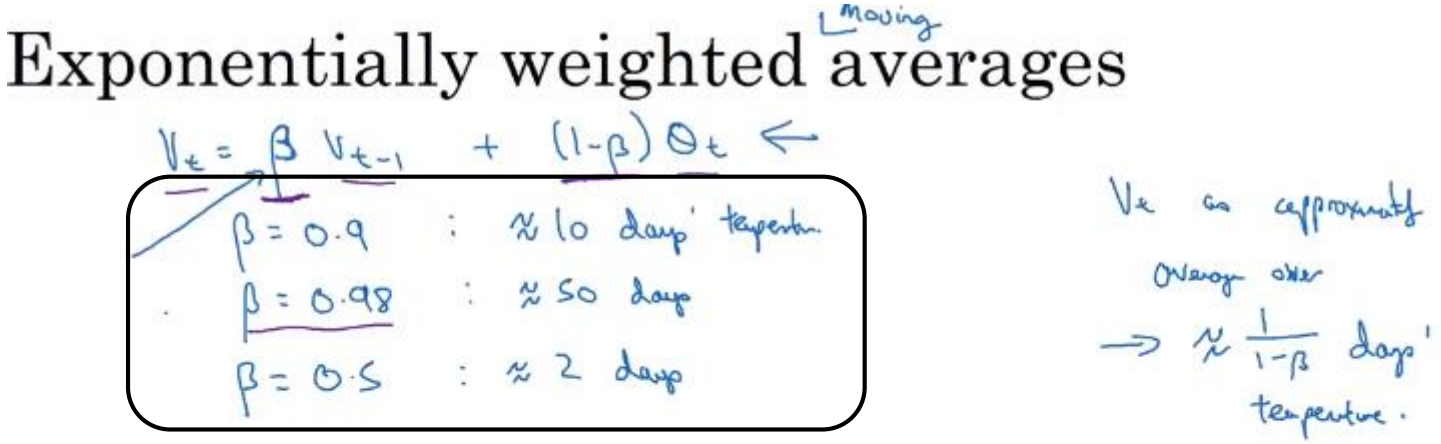
Exponentially weighted averages (지수 가중 이동 평균)



Vt를 계산한 값은 대략적으로 1/1-베타 곱하기 일별 기온의 평균과 같음

Andrew Ng

Exponentially weighted averages (지수 가중 이동 평균)



베타가 0.9면 이는 10일 동안의 기온의 평균과 같음. 베타가 0.98이면 50일 기온의 평균과 거의 같음
베타 값이 클수록 선이 부드러워짐. 더 많은 날짜의 기온의 평균을 이용하기 때문에 곡선이 부드러워짐

또한 곡선의 올바른 값에서 어 더 멀어짐 더 큰 범위에서 기온을 평균하기 때문에
기온이 바뀔 경우 지수가중평균 공식은 더 느리게 적용
베타가 0.98이면 이전 값에 더 많은 가중치를 주고 현재의 기온에는 작은 가중치를 주게 됨
기온이 올라가거나 내려가면, 지수가중평균은 베타가 커서 더 느리게 적용
베타가 0.5라면 2일간의 기온만 평균, 더 노이즈가 많고 이상치에 민감, 기온 변화에 빠르게 적용
베타 하이퍼파라미터의 값을 바꿈으로써 약간씩 다른 효과를 줄 수 있다

Understanding exponentially weighted averages

Exponentially weighted averages

$$v_t = \beta_{v_t-1} + (1-\beta)\theta_t$$
$$v_{100} = 0.9v_{99} + 0.1\theta_{100}$$
$$v_{99} = 0.9v_{98} + 0.1\theta_{99}$$
$$V_{100} = 0.9v_{99} + 0.1\theta_{100}$$
$$= 0.1\theta_{100} + 0.1 \times 0.9 \cdot \theta_{99} + 0.1 \cdot (0.9)^2\theta_{98} + 0.1 \cdot (0.9)^2$$

Vt = 베타 * Vt-1 + (1-베타) * 세타t

V100 = 0.9 * V99 + 0.1 * 세타100
V99 = 0.9*V98 + 0.1 * 세타99
…
V100 = 0.1*세타100 + 0.9*0.1*세타99 + 0.1*(0.9)^2*세타98+ …

V100을 구하는 과정은 이 두 함수 간에 요소별 곱셈을 해서 더하는 것
일일 온도에 지수적으로 감소하는 함수를 곱해주고 모두 더함
앞에 곱해지는 계수들은 모두 더하면 1 또는 1에 가까운 값이 됨

Understanding exponentially weighted averages

온도가 1/3이 될 때 까지 약 10일이 걸림
이런 이유 때문에 베타가 0.9와 같을 때 지난 10일간의 온도에만 초점을 맞춰 가중평균을 계산한다면,
10일 뒤에는 가중치가 현재 날짜의 가중치의 1/3으로 줄어듭니다

그와 반대로 베타가 0.98이라면 처음 50일 동안의 1/e보다 가중치는 더 커질 것이다
따라서 50일의 온도의 평균은 더 급격히 빠르다

Implementing exponentially weighted averages

온도가 1/3이 될 때 까지 약 10일이 걸림
이런 이유 때문에 베타가 0.9와 같을 때 지난 10일간의 온도에만 초점을 맞춰 가중평균을 계산한다면,
10일 뒤에는 가중치가 현재 날짜의 가중치의 1/3으로 줄어듭니다

그와 반대로 베타가 0.98이라면 처음 50일 동안의 1/e보다 가중치는 더 커질 것이다
따라서 50일의 온도의 평균은 더 급격히 빠르다

Implementing exponentially weighted averages

지수평균을 얻는 식의 장점은 아주 적은 메모리를 사용한다는 것
VΘ 이 실수 하나만을 컴퓨터 메모리에 저장하고 가장 최근에 얻은 값을 이 식에 기초하여 덮어쓰기만 하면 됨

- V = Beta × V + (1-Beta) x Theta

메모리 절약 가능함.

Bias correction

$$\frac{V_t}{1-\beta^t}$$

$$t = 2; 1 - \beta^t = 1 - (0 - 98)^2 = 0.396$$

Gradient descent with momentem

Implementing exponentially weighted averages

평균을 계산하는 가장 정확하고 최선의 방법은 아님
지난 10일 또는 50일 간의 온도를 더하고 10이나 50으로 나누는 것이 더 나은 추정치를 제공함
(그러나 이렇게 하면더 많은 메모리가 필요, 더 복잡한 구현!)

많은 변수의 평균을 계산하는 경우 컴퓨터의 계싼 비용과 메모리 효율 측면에서 더 비효율적

Bias correction In exponentally weighted average

편향보정이라고 불리는 기술적인 세부사항으로 평균을 더 정확하게 계산 가능

베타가 크면 v1의 값이 훨씬 더 낮아져서 첫번째 날의 온도를 잘 추정할 수 없다

V2 = 0.98 x 0.02 x 세타1 + 0.02 x 세타2
한 해의 첫 두 날짜를 추정한 값이 좋지 않은 추정이 됨
추정의 초기 단계에서 더 정확하게 보정

- V = Beta × V + (1-Beta) x Theta

메모리 절약 가능함.

Bias correction

$$\frac{V_t}{1-\beta^t}$$

Vt를 취하는 대신에 이걸 취함 t가 더 커질수록 베타t는 0에 가까워짐
따라서 t가 충분히 커지면 편향보정 효과가 거의 없어짐

$t = 2; 1$

Gradient descent with momentem

초기 단계 학습에서 편향보정은 더 나은 온도의 추정값을 얻을 수 있도록 도와줌

Implementing exponentially weighted averages

머신러닝에서 지수가중평균을 구현하는 대부분의 경우는

사람들이 편향 보정을 거의 구현하지 않는다

왜냐하면 초기 단계를 그냥 기다리고 편향된 추정이 지나간 후부터 시작한다

그러나 초기 단계의 편향이 신경쓰인다면 편향보정은 초기에 더 나은 추정을 얻는데 도움을 줄 것이다

Gradient decent with momentum

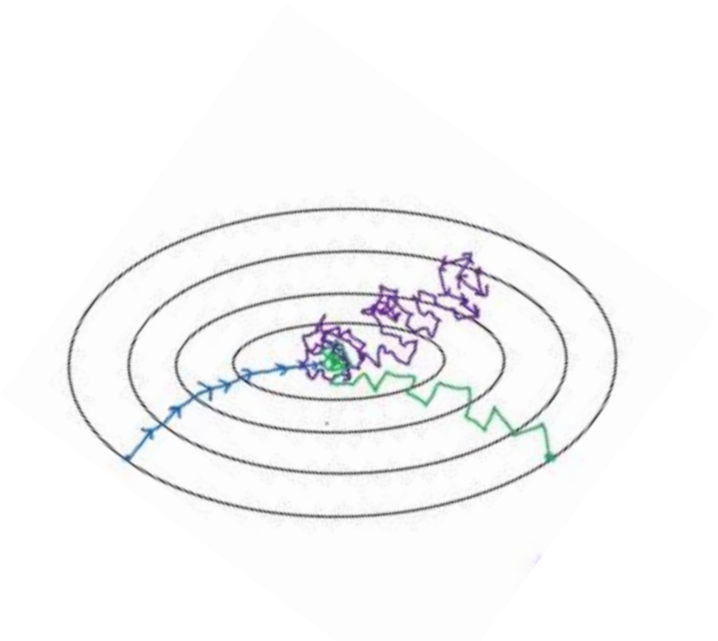
β 가 나오는데 이값에 전단계의 gradient값을 곱하고 (1-β)에 이번 gradient를 곱해서 이전 값을 통해 현재 값이 너무 튀는걸 어느정도 보정하고 현재값도 어느정도 gradient의 움직임에 영향을 미칠 수 있도록 하는 방법을 momentum이라고 하고 이 방법은 exponentially weighted moving averages을 사용했다.

Gradient decent with momentum

위 아래로 일어나는 이런 진동은 경사하강법의 속도를 느리게하고, 더 큰 학습률을 사용하는 것을 막는다

왜냐하면 오버슈팅하게 되어 발산할 수 있기 때문

따라서 학습률이 너무 크지 않아야 진동이 커지는 것을 막을 수 있다



수직축에서도 진동을 막기 위해 학습이 더 느리게 일어나기를 바라지만

수평축에서는 더 빠른 학습을 원한다

Gradient decent with momentum

Vdw = βVdw + (1- β)dw : 이동평균을 w에 대한 도함수로 계산

vΘ = βV + (1-Θt)

Vdb = β *V *db + (1-β)db

w = w-$\propto$Vdw : w를 사용해 가중치 업데이트

b-= b- $\propto$Vdb

이것은 경사하강법의 단계를 부드럽게 만들어 줌

지그재그 의 경사 평균을 구하면 수직 방향의 진동이 0에 가까운 값으로 평균이 만들어짐

Gradient decent with momentum

진행을 늦추고 싶은 수직방향에서는 양수와 음수를 평균하기 때문에 평균이 0이 됨

반면 수평방향에서는 모든 도함수는 오른쪽을 가리키고 있기 때문에

수평 방향의 평균은 꽤 큰 값을 가짐

Gradient decent with momentum

여기서 경사하강법은 수직 방향에서는 훨씬 더 작은 진동이 있고

수평 방향에서는 더 빠르게 움직인다는 것을 찾을 수 있음

따라서 이 알고리즘은 더 직선의 길을 가거나 진동을 줄일 수 있게 함

이 모멘텀에서 얻을 수 있는 직관은 bowl(그릇) 모양의 함수를 최소화하려면

도함수의 항들은 아래로 내려갈 때 가속을 제공

Vdw = βVdw + (1- β)dw

Vdb = β *V *db + (1-β)db

도함수 항들에게 가속을 제공해야함

그리고 이 모멘텀 항들은 속도를 나타낸다고 볼 수 있음

베타의 값은 1보다 조금 작기 때문에 마찰을 제공해서 공이 제한없이 빨라지는 것을 막음

Gradient decent with momentum

경사하강법이 모든 이전 단계를 독립적으로 취하는 대신에

그릇에 내려가는 공에 가속을 주고 모멘텀을 제공할 수 있음

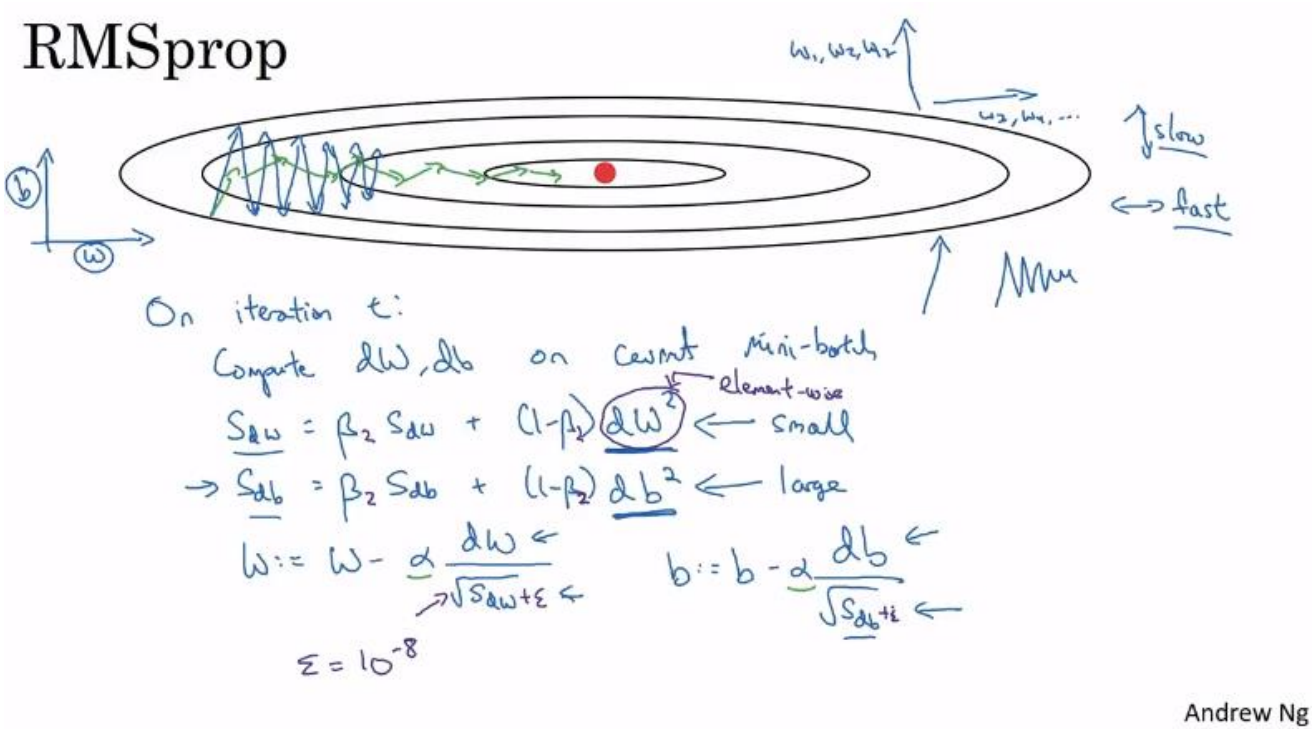
학습률 알파와 지수가중평균을 제어하는 베타라는 두가지 하이퍼파라미터가 있음

RMSprop

경사하강법에서 수평 방향으로 진행을 시도해도 많은 진동이 있음

수직방향의 학습속도를 낮추기 하기 위한 것이고 수평 방향의 속도를 빠르게 하기 위한 것

반복 t에서 현재의 미니배치에 대한 보통의 도함수 dw와 db를 계산할 것!

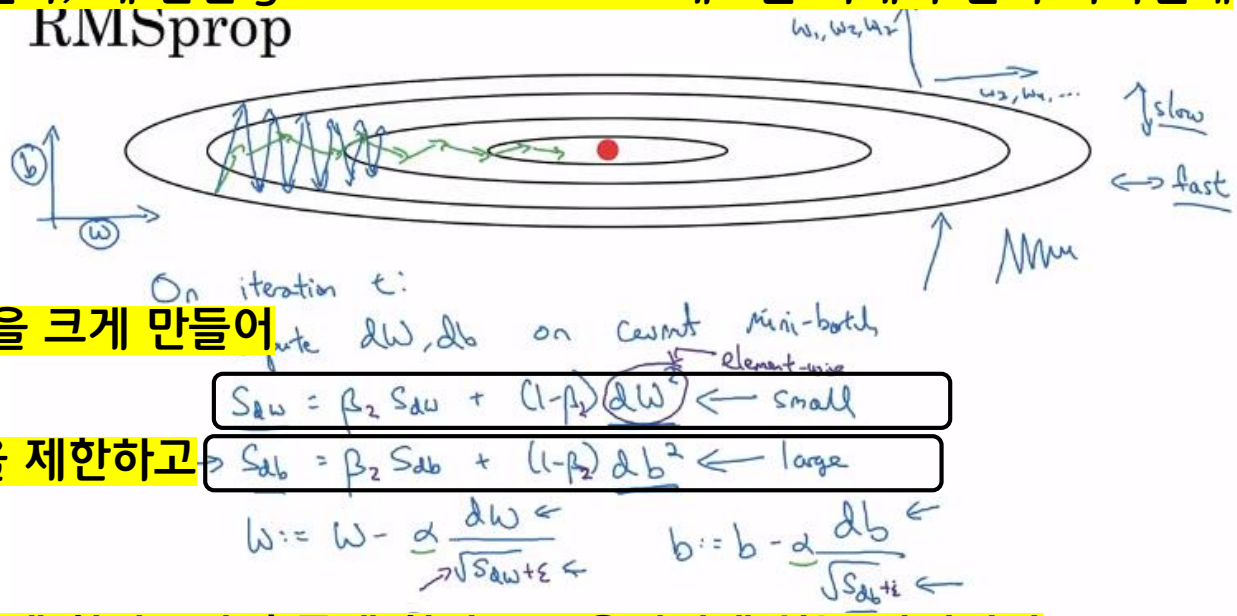


Andrew Ng

RMSprop

Mini-Batch Gradient를 사용할때 속도를 빠를게 할 수 있는 다른 방법이며 기본적으론 momentum의 아이디어와 유사하다. 차이점이라면 momentum의 경우 β값에 의해 보정된 새로운 gradient값=Vdw을 파라미터 W를 업데이트 하는는데 바로 사용했다면, RMSprop의 새로운 gradient값 Sdw = 기본 gradient값을 제곱해서 사용하는고 (아래 그림의 보라색 동그리마) , gradient를 update할때 새로운 gradient값 Sdw에 루트 씌운 값으로 나눈다.

파라미터 W (행축) 와 Bias B (열축) 에 관한 gradient descent 그래프를 아래와 같이 나타낼때



Sdw의 값을 작게해서 행축 방향의 움직임을 크게 만들어

Sdb 값을 크게해서 열축방향의 움직임을 제한하고

파란색 화살표가 초록색 화살표로 움직이게 하는 방법이다

Andrew Ng

RMSprop

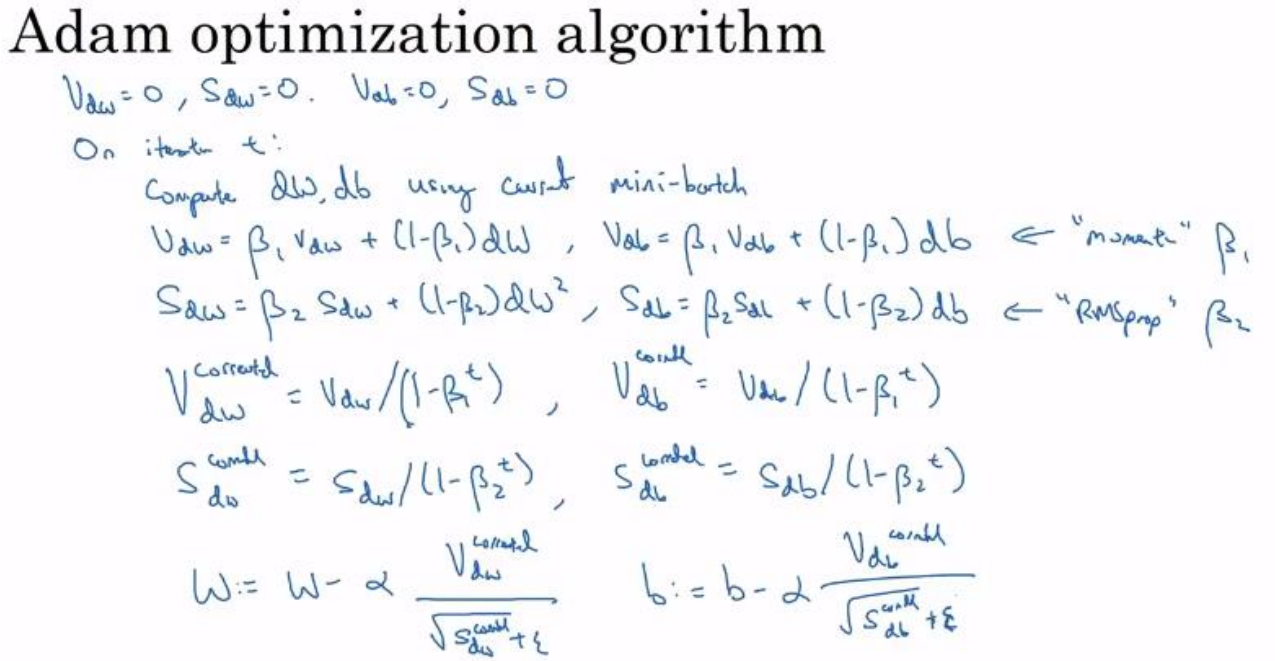
이것의 효과는 큰 학습률을 사용해 빠르게 학습하고 수직 방향으로 발산하지 않는다
실제로는 매개변수의 고차원 공간에 있기 때문에 진동을 줄이려는 수직 차원은
w1,w2…..w17의 매개변수 집합이고, 수평방향의 차원은 w3,w4…처럼 나타날 것이다
따라서 w와 b의 분리는 표현을 위한 것이고 실제로 dw와 db는 매개변수 벡터이다

RMSprop는 진동을 줄이는 효과가 있다는 점에서 모멘텀과 비슷하며,
더 큰 학습률을 사용할 수 있게 해서 속도를 올려줌

Adam optimization algorithm

RMSprop + 모멘텀

Momentum과 RMSprop를 섞어서 학습을 빠르게 만드는 알고리즘이다. W와 Bias를 업데이트 할때 두가지 모두를 반영한다. 원래 가장 기초는 backpropagation을 통해 나온 dw와 db를 곱하는 거였다면 여기서는 momentum으로 구한 새로운 gradient Vdw, Vdb를 분모로 사용하고 RMSprop로 구한 새로운 gradient를 분모로 사용해서 learning rate α값에 곱해 계산한다.

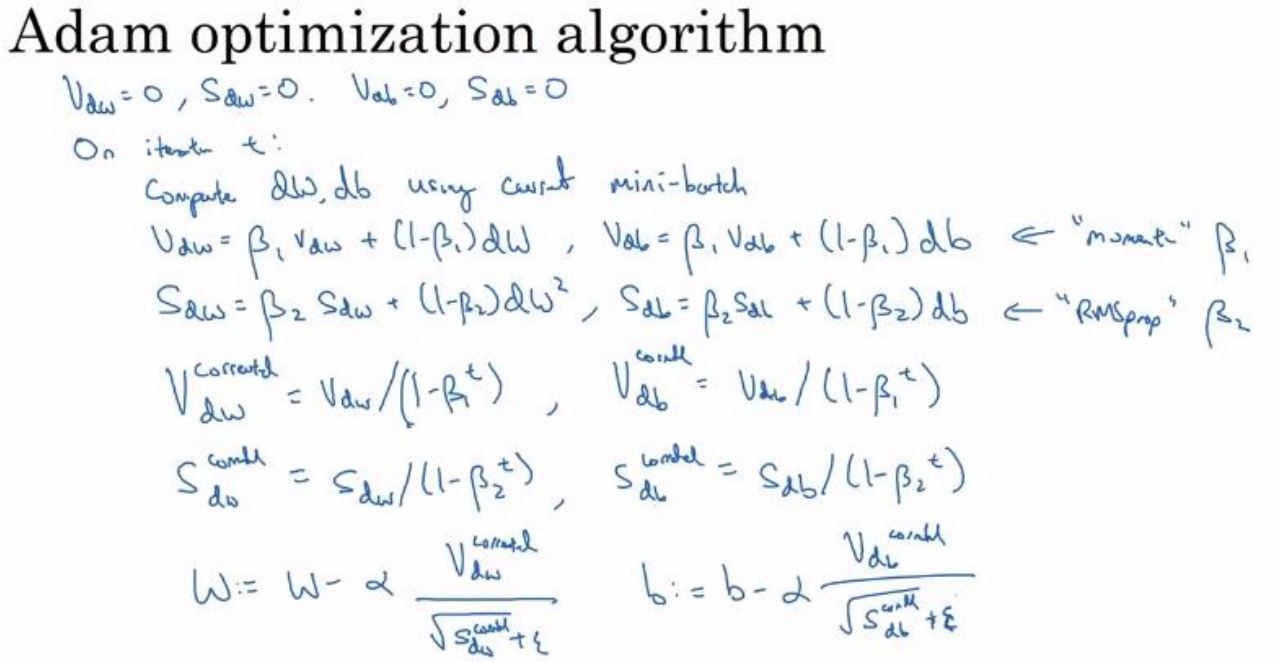


Andrew Ng

Adam optimization algorithm

매우 넓은 범위의 아키텍처를 가진 서로 다른 신경망에 잘 작동한다는 것이 증명된

일반적으로 많이 쓰이는 학습 알고리즘



Andrew Ng

Hyperparameters choice

따라서 이 알고리즘은 많은 하이퍼파라미터가 있다

학습률 알파 : 다양한 값 시도 매우 중요!!!

베타1 : 0.9 (dw의 이동평균, 가중평균, 모멘텀에 관한 항)

기본적인 값으로 보통 0.9를 사용

베타2 : 0.999 – Adam 저자가 추천하는 값 (dw^2) 크게 상관은 없다고 함

앱실론 : 논문 저자는 10^-8추천

다른건 잘 보정 안하고 학습률만 좀 보정함

Hyperparameters choice

Adam : Adaptive moment estimation

베타1이 도함수의 평균을 계산하므로 첫번째 모멘트이고

베타2가 지수가중평균의 제곱을 계산하므로 두번째 모멘트

adam 최적화 알고리즘 신경망을 더 빠르게 작동하게 함!

Learning rate decay

학습알고리즘의 속도를 높이는 한가지 방법은 시간에 따라 학습률을 감쇠시키는 것!

상당히 작은 미니배치에 대해 경사하강법을 구현한다고 가정
약간의 노이즈는 있지만 최소값으로 향하는 경향을 보일 것!
정확하게 수렴은 안하지만 주변을 돌아다님
왜냐하면 고정된 값인 알파를 사용했고, 서로 다른 미니배치에 노이즈가 있기 때문

천천히 학습률 알파를 줄이면, 알파가 여전히 큰 초기 단계에서는 상대적으로 빠른 학습이 가능
그러나 알파가 작아지면 단계마다 진행 정도가 작아지고 최솟값 주면에 밀집된 영역에서 진동함
훈련이 계속되도 최솟값 주변에 배회하는 대신에,
따라서 알파를 천천히 줄이는 것의 의미는 학습 초기 단계에서는 훨씬 큰 스텝으로 진행하고
학습이 수렴할수록 학습률이 느려져 작은 스텝으로 진행. 학습률 감쇠를 구현하는 방법.

Learning rate decay

이제까지 언급한 부분중에서 learning late α값을 건든적이 없는데 이거 잘생각해보면 엄청 쉽게 학습을 빨리 하게 할수있다. 아래 그림처럼 learning rate α값을 특정 값으로 설정 하면 학습을 진행할 동안 α에 의해 움직이는 양은 정답에 가까울 수록 줄어들 수는 있겠지만(Loss가 작아지므로) 정답을 앞에두고도 움직이는 양이 상대적으로 커서 정답을 못찾고 주변을 크게 우회할 경우가 생긴다 이말을 바꿔 말하면 정답 근처에서도 오차가 꽤 크게 날수 있다는건데, 아래 그림 파란색 선처럼 말이다. 하지만 우리는 위에 설명한대로 학습 중 각 에폭 마다 α값을 일정 **비율**로 줄여주면 처음엔 엄청 확확 움직이고 나중에 정답에 가까워질때즘엔 천천히 움직여서 아래 그림의 초록색 선처럼 정답 근처에 머물게 만들수 있을거다. 이렇게 비율로 움직이는걸 learning rate decay라고 한다.