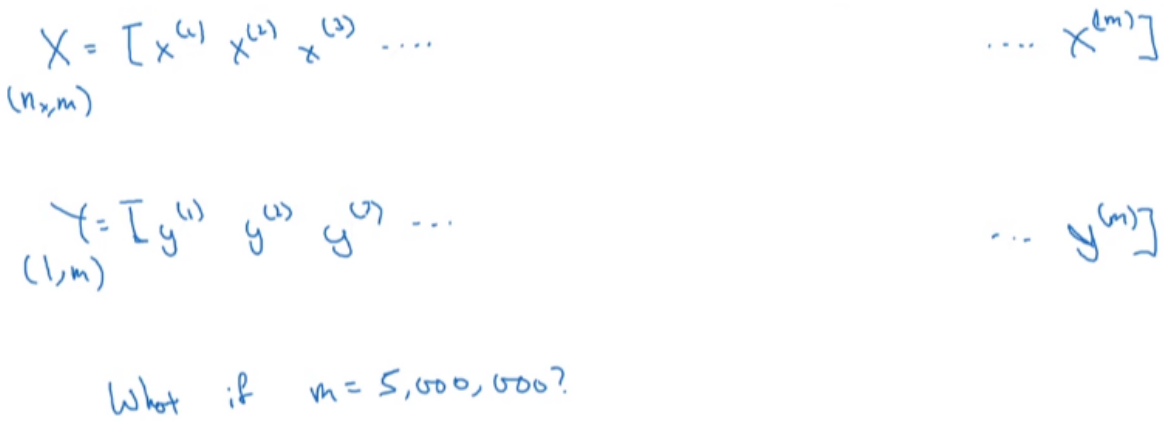
미니 배치 경사하강법

머신러닝을 적용하는 것은 잘 작동되는 모델을 찾기 위해 많은 훈련을 거쳐야 하는 반복적인 과정으로 매우 실험적이다. 따라서 모델을 빠르게 훈련시키는 것이 매우 중요한다. 큰 데이터 세트에서 신경망을 훈련시킬 수 있을 때 딥러닝이 빅데이터에서 가장 잘 작동된다는 것도 훈련을 어렵게 만든다. 큰 데이터 세트에서 훈련하는 것은 매우 느린 과정이다. 따라서 좋은 최적화 알고리즘을 찾는 것은 효율성을 좋게 만들어준다.

Batch vs. mini batch gradient descent

Vectorization allows you to efficintly compute on *m* examles. : 벡터화가 m 개의 샘플에 대한 계산을 효율적으로 만들어 준다. 명시적인 반복문 없이도 훈려세트를 진행할 수 있도록 한다. 따라서 훈련샘플을 받아서 큰 행렬에 저장한다.



- **배치 경사 하강법** : 전체 훈련 샘플에 대해 훈련 후 경사 하강법을 구현 -> 경사 하강법의 작은 한단계를 밟기 전에 모든 훈련세트를 처리해야 함. 또 경사하강법의 다음 단계를 밟기 전에 다시 오백만개의 전체 훈련샘플을 처리해야함.

=> 따라서 오백만 개의 거대한 훈련샘플을 모두 처리하기 전에 경사하강법이 진행되도록 하면 더 빠른 알고리즘을 얻을 수 있다.

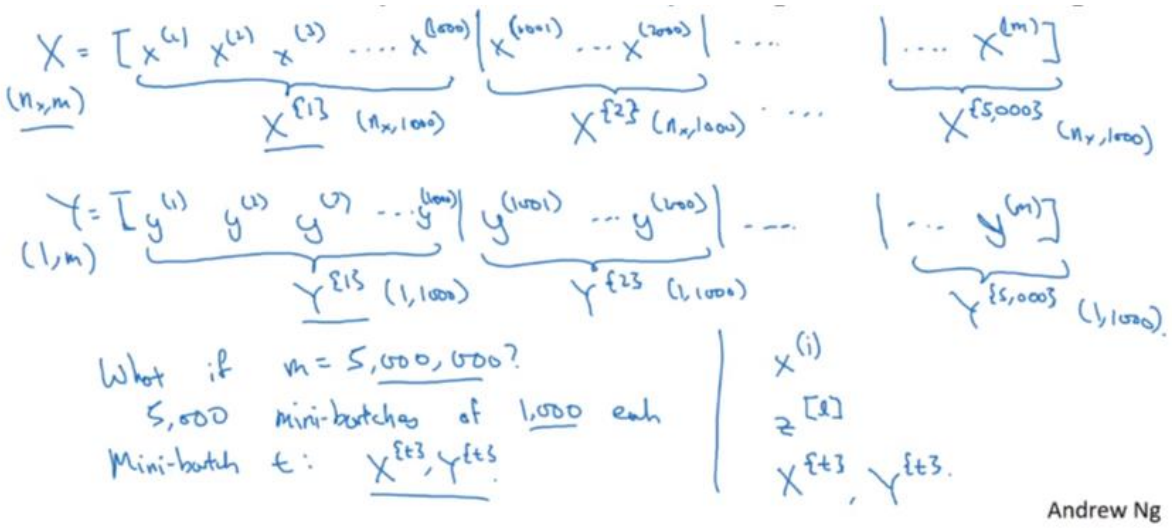
- **미니 배치 경사 하강법** : 전체 훈련 샘플을 작은 훈련세트들로 나누고(이를 미니배치라고 함), **미니배치 훈련후 경사하강법을 진행**

예를 들어 전체 훈련세트가 5,000,000 일 때, 각각의 미니배치가 1,000 개의 샘플을 갖는다고(사이즈가 1,000 인 미니배치 5,000 개) 가정하고 훈련 및 경사 하강법을 진행한다. 미니배치 경사하강법의 새로운 **표현법**은 다음과 같다.

- i 번째 훈련 세트 : x^(i)

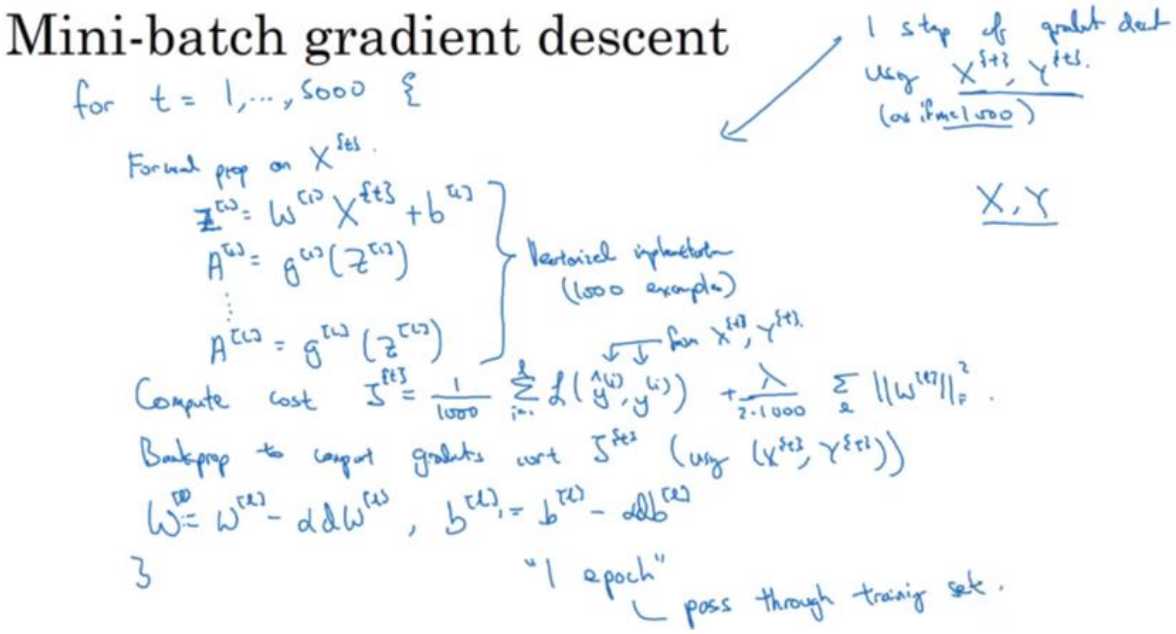
- l 번째 신경망의 z 값 : z^[l]

- t 번째 미니배치 : X^6^{t},Y^{t}



Mini-batch gradient descent

훈련세트에서 미니배치 경사 하강법을 실행하기 위해 t=1... 5,000 반복문을 돌린다. 반복문 안에서는 한단계의 경사하강법을 구현한다.

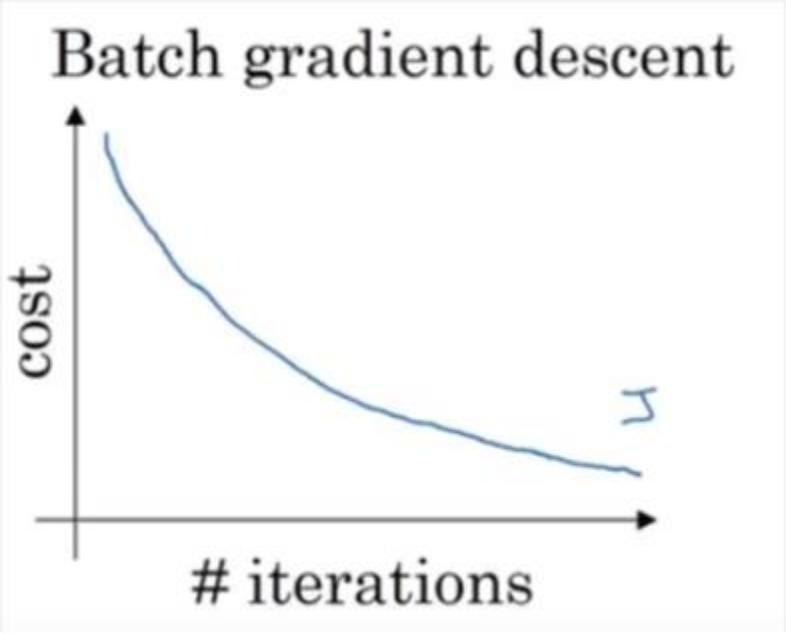


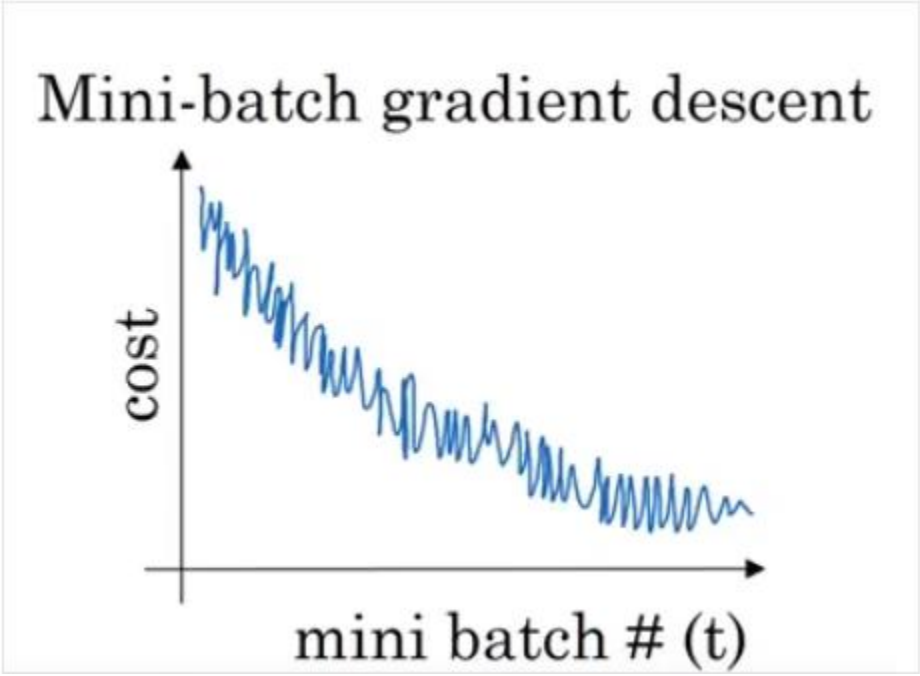
1 epoch(에포크) : 훈련세트를 거치는 한 반복을 의미

따라서 배치 경사하강법에서 훈려세트를 거치는 한 반복은 오직 하나의 경사하강 단계만을 할 수 있게 한다. **미니배치 경사하강법의 경우 훈련 세트를 거치는 한 반복은 5,000 개의 경사 하강 단계를 거치도록 한다.**

미니 배치 경사하강법 이해하기

Training with mini batch gradient descent





- **배치 경사 하강법** : 모든 반복에서 전체 훈련세트를 진행하고 **각 반복마다 비용이 감소해야 한다.**

- **미니 배치 경사 하강법** : 모든 반복마다 감소하지 않는다. 전체적으로는 **비용함수가 감소하는 경향**을 보이지만 **노이즈가 많이 발생**한다. (+) 노이즈가 발생하는 이유는 X^{1}과 Y^{1}이 상대적으로 쉬운 미니배치라서 비용이 약간 낮은데 우연적으로 X^{2}와 Y^{2}가 더 어려운 미니배치라서 등의 이유로 비용이 약간 더 높아질 수 있다.

Choosing our mini-batch size

우리가 선택해야 하는 매개변수 중 하나는 **미니배치의 크기**이다.

i) m 이 훈련세트 크기일 때, 미니배치 크기가 m 과 같은 경우 = 배치 경사 하강법

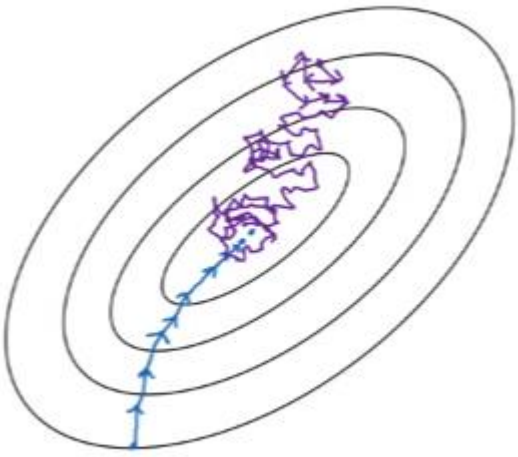
이 경우에는 하나의 미니배치만을 갖게 되고 이 미니배치의 크기는 전체 훈련 세트와 같다. 따라서 미니배치 크기를 m 으로 설정하는 것은 일반적인 경사 하강법과 같다.

ii) 미니배치 크기 = 1 인 경우; 확률적 경사 하강법

각각의 샘플은 하나의 미니배치이다. 첫번째 미니 배치인 X^{1}과 Y^{1}을 살펴보면 미니배치 크기가 1 일 때 이는 첫번째 훈련샘플과 같다. 즉, 첫번째 훈련샘플로 경사 하강법을 하는 것이다. 다음에 두번째 미니배치를 살펴보면 이는 두번째 훈련샘플과 같고 이에 대한 경사하강단계를 취하게 된다. ... 이런 식으로 한번에 하나의 훈련샘플만을 살펴보며 계속 진행하는 것이다.

=> i), ii)의 두가지 극단적인 경우가 비용함수를 최적화할 때 무엇을 하는지 살펴보자.

i)은 파란색 화살표의 경로를 따르고 ii)는 보라색 화살표의 경로를 따른다.



iii) 실제로 우리가 사용하는 미니배치 크기는 1 과 m 사이일 것

(1 은 상대적으로 작고 m 은 상대적으로 큰 값이다)

- 배치 경사 하강법; 미니배치의 크기는 m 과 같다. -> 매우 큰 훈련 세트를 모든 반복에서 진행하게 됨 -> 단점) 한 반복에서 너무 오랜 시간이 걸림

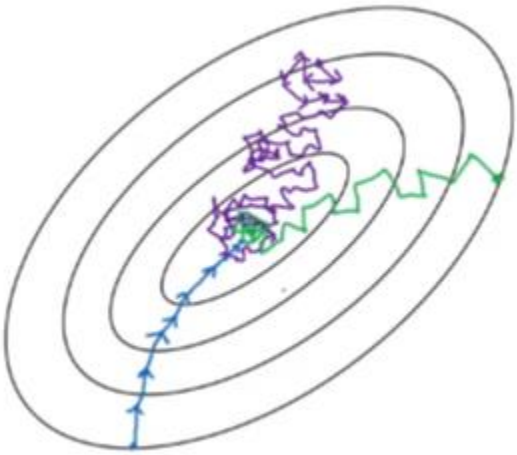
- 확률적 경사 하강법; 하나의 샘플만 처리한 뒤에 계속 진행 가능해 매우 간단, 노이즈도 작은 학습률을 사용해 줄일 수 있음 -> 단점) 벡터화에서 얻을 수 있는 속도 향상을 잃게 됨 - 한 번에 하나의 훈련세트를 진행하기 때문에 각 샘플을 진행하는 방식이 매우 비효율적임

따라서 가장 잘 작동하는 값은 1 과 m 사이의 있는 값이다. 실제로 이는 가장 빠른 학습을 제공한다. 이를 통한 **두가지 장점**이 있다.

1. 많은 벡터화를 얻음 : 미니배치의 크기가 1000 개의 샘플이라면 1000 개의 샘플에 벡터화를 하게 된다. 그렇다면 한 번에 샘플을 진행하는 속도가 더 빨라지게 된다.

2. 전체 훈련세트가 진행되기를 기다리지 않고 진행가능 : 각각의 훈련세트의 에포크는 5000 번의 경사 하강 단계를 허용한다.

미니배치는 항상 최솟값으로 수렴한다고 보장할 수는 없지만 더 일관되게 전역의 최솟값으로 향하는 경향이 있다. 그리고 매우 작은 영역에서 항상 정확하게 수렴하거나 진동하게 된다. (iii)는 초록색 화살표를 따른다)



그렇다면 미니배치 크기가 m 이나 1 이 아닌 그 사이의 값이어야 한다면 그 값을 어떻게 선택할까?

1. 만약 훈련세트가 작다면(2000 개 이하) 모든 훈련세트를 한번에 학습시키는 배치 경사하강을 진행한다.

2. 이와 달리 2000 개 이상의 훈련세트가 클 경우 전형적으로 선택하는 미니배치 사이즈는 64, 128, 256, 512 와 같은 2 의 제곱수이다.

+ 미니배치에서 모든 X^{t}와 Y^{t}가 CPU 와 GPU 메모리에 맞는지 확인해라. 이는 여러분의 애플리케이션과 하나의 훈련샘플의 크기에 달려있다. 그렇지 않으면 성능이 갑자기 떨어지고 훨씬 나빠지게 된다.

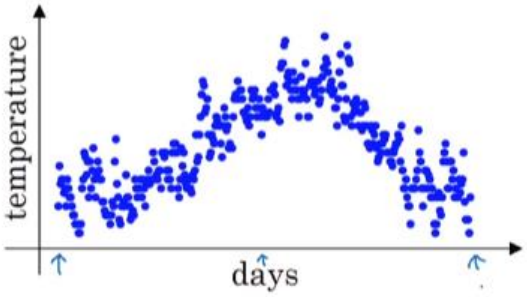
미니배치 크기는 빠른 탐색을 통해 찾아내야 하는 또 다른 하이퍼파라미터이다. 가장 효율적이면서 비용함수 J 를 줄이는 값을 찾아내야 한다.

지수 가중 이동 평균

경사하강법 및 미니배치 경사하강법보다 더 효율적인 알고리즘을 이해하기위해 **지수 가중 이동 평균**을 먼저 이해해야 한다.

Temperature in London

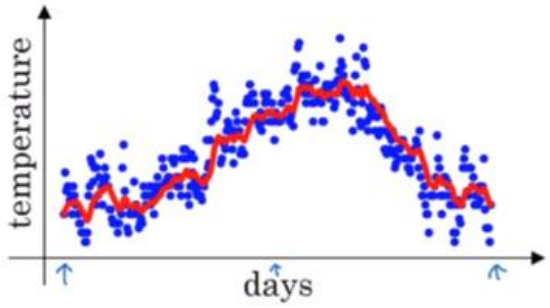
$\theta_1 = 40°F$ 4°C
$\theta_2 = 49°F$ 9°C
$\theta_3 = 45°F$ ⋮
⋮
$\theta_{180} = 60°F$ 15°C
$\theta_{181} = 56°F$ ⋮
⋮



이 그래프에서 약간의 노이즈가 있지만 지역 평균이나 이동 평균의 흐름을 계산하고 싶다면 이런 방법이 있다.

v_0 를 0 으로 초기화한다. 매일 이전의 값에 0.9 를 곱하고 여기에 0.1 곱하기 해당 날의 기온을 더해줄 것이다. (θ1 : 첫 번째 날의 기온) .. 이런 식으로 계속 한다. 따라서 일반적인 식은 V_t = 0.9*V_(t-1) + 0.1*θ_t 가 되고 그래프는 다음과 같아진다. **일별 기온의 지수가중평균**을 얻게 된다.

$\theta_1 = 40°F$ 4°C ←
$\theta_2 = 49°F$ 9°C
$\theta_3 = 45°F$ ⋮
⋮
$\theta_{180} = 60°F$ 15°C
$\theta_{181} = 56°F$ ⋮
⋮



$V_0 = 0$
$V_1 = 0.9\, V_0 + 0.1\, \theta_1$
$V_2 = 0.9\, V_1 + 0.1\, \theta_2$
$V_3 = 0.9\, V_2 + 0.1\, \theta_3$
⋮
$V_t = 0.9\, V_{t-1} + 0.1\, \theta_t$

Exponentially weighted averages

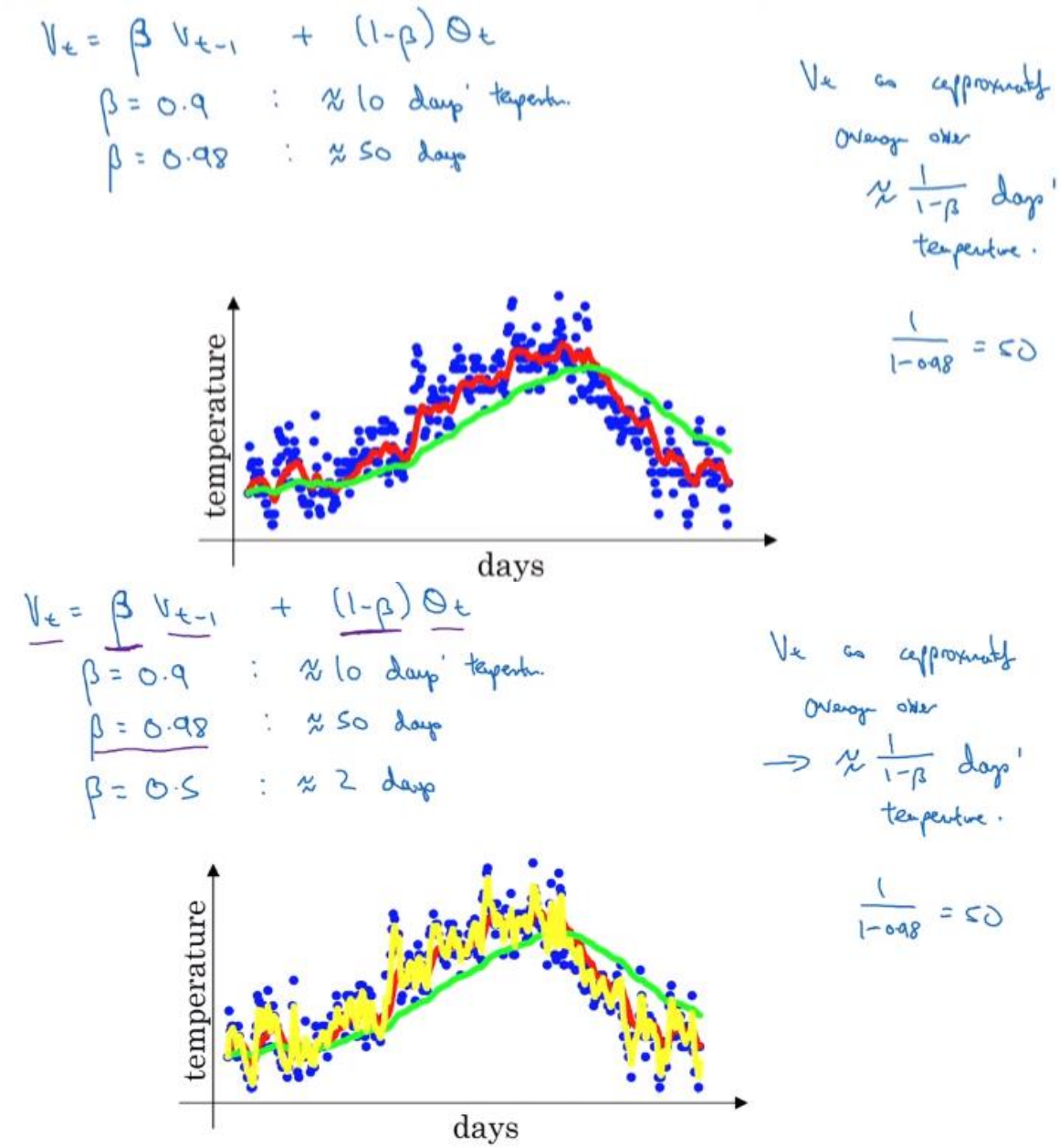
공식을 다시 살펴보자. (β=0.9) **V_t 를 계산한 값은 대략적으로 1/(1-β) 곱하기 일별 기온의 평균과 같다.**

i) **β=0.9** 와 같은 경우 이는 10 일 동안 기온의 평균과 같다. 그것이 그래프의 붉은색 선이다.

ii) β가 1 과 매우 가까운 경우(**β=0.98**); 1/(1-0.98)은 50 과 비슷하므로 이 값은 기온의 평균과 거의 같다. 이는 그래프의 초록색 선과 같다.

iii) **β=0.5**; 공식에 의해서 2 일의 기온만 평균하는 것과 같다. 오직 2 일만의 기온을 사용했기에 더 노이즈가 많고 이상치에 더 민감하다. 그러나 기온 변화에 더 빠르게 적응한다. 이는 그래프의 노란색 선과 같다.

여기서 **β값이 클수록** 더 많은 날짜의 기온의 평균을 이용하기 때문에 **곡선이 더 부드러워짐**을 알 수 있다. 하지만 더 큰 범위에서 기온을 평균하기 때문에 **곡선이 올바른 값에서 더 멀어진다.** 그래서 기온이 바뀔 경우에 지수가중편균 공식은 더 느리게 적응한다. 따라서 지연되는 시간이 더 크다.



따라서 이 공식은 지수가중평균을 구현하기 위한 공식이다. 통계학에서는 지수가중이동평균이라고 부른다. (줄여서 지수가중평균) 이 매개변수 혹은 학습 알고리즘의 하이퍼파라미터의 값을 바꿈으로써 약간씩 다른 효과를 얻게 되고 이를 통해 가장 잘 작동하는 값(빨간색 곡선이 주는 값)을 찾게 된다. **빨간색 곡선이 초록색이나 노란색 곡선보다는 더 나은 평균을 제공한다.**

지수 가중 이동 평균 이해하기

지수가중평균은 **신경망 훈련에 사용하는 몇 가지 최적화 알고리즘의 주요 구성요소**가 될 것이다. 이 알고리즘이 하는 일에 대해서 깊게 알아보자.

Exponentially weighted averages

아래 공식은 지수가중평균을 구현하는데 주요한 공식이다. 일일 기온의 평균을 계산하는 방법에 대해 수학적으로 더 알아보자.

β=0.9 로 설정하고 앞서 나온 지수가중이동평균 식을 하나의 값으로 표현하게 되면 다음과 같다.

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$$v_{100} = 0.9v_{99} + 0.1\theta_{100}$$
$$v_{99} = 0.9v_{98} + 0.1\theta_{99}$$
$$v_{98} = 0.9v_{97} + 0.1\theta_{98}$$
$$\cdots$$
$$V_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9v_{97}))$$
$$= 0.1\theta_{100} + 0.1\times0.9\cdot\theta_{99} + 0.1(0.9)^2\theta_{98} + 0.1(0.9)^3\theta_{97} + \cdots$$

v100=0.1θ100+0.1×0.9θ99+0.1×(0.9)2θ98+⋯ V_100 은 다음과 같이 표현될 수 있으며 이것은 가중치의 합(즉, θ_100 의 가중치의 평균)이다.

이를 그림으로 표현하면 **지수적으로 감소하는 그래프**가 된다. 왜냐하면 v100 을 구하는 과정에서 각 요소별 곱셉(0.1 x (0.9)^n)을 해서 더하기 때문이다.

위의 식들에서 앞에 곱해지는 계수들을 모두 더하면 1 또는 1 에 가까운 값이 되는데 이는 편향 보정이라고 불린다. (아래에서 설명) 이들에 의해 지수가중평균이 된다. 그렇다면 **얼마의 기간이 이동하면서 평균이 구해지는걸까?**

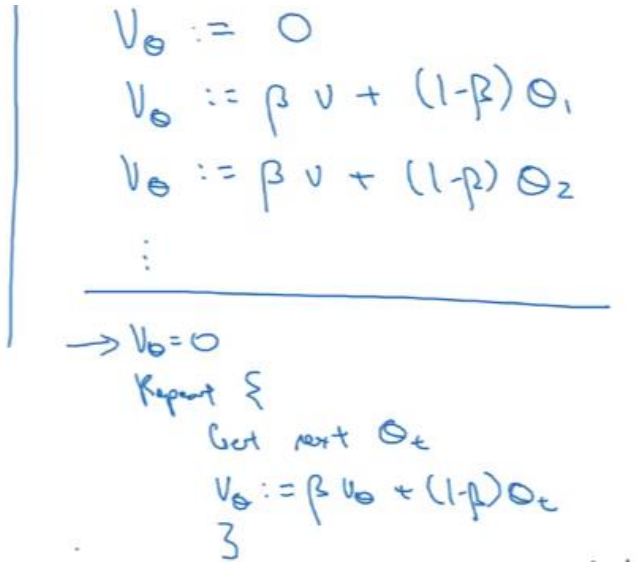
β가 0.9 와 같을때, 지난 10 일간의 온도에만 초첨을 맞춰 가중평균을 계산한다면 10 일 뒤에는 현재 날짜의 가중치의 1/3 으로 줄어들게 된다. 반대로 β가 0.98 이라면 0.98^50 이 대략적으로 1/e 와 같다. 처음 50 일 동안의 1/e 보다 가중치는 더 커질 것으로 보인다. 감소는 가프르게 일어날 것이다. 따라서 직관적으로 50 일의 온도의 평균은 더 급격히 빠르다. 왜냐하면 ε은

0.02 이기 때문이다. 따라서 1/ε 은 50 과 같다. 이는 1/(1-β)와도 대략적으로 같다. 평균적인 온도가 몇 일 정도가 될지에 관한 상수를 알려준다. 이는 관습적으로 쓰이는 것이지 수학공식은 아님에 유의하자.

- β=(1−ε) 라고 정의 하면
- (1−ε)^n = 1/e 를 만족하는 n 이 그 기간이 되는데, 보통 1/ε 으로 구할 수 있다.

Implementing exponentially weighter averages

$$v_0 = 0$$
$$v_1 = \beta v_0 + (1-\beta)\,\theta_1$$
$$v_2 = \beta v_1 + (1-\beta)\,\theta_2$$
$$v_3 = \beta v_2 + (1-\beta)\,\theta_3$$
$$\dots$$



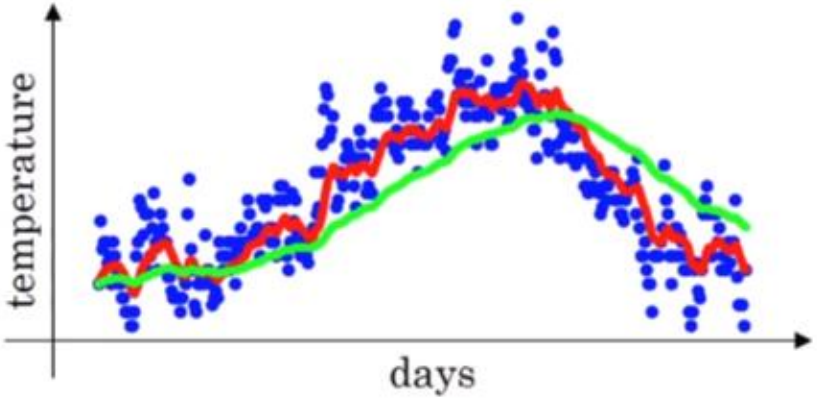
실제로 구현할 때는 v 를 0 으로 초기화한다. 가끔 v 에 아래 첨자 θ를 한 표기법을 사용하기도 한다. v 가 θ를 매개변수로 하는 지수가중평균을 계산하는 것을 나타내기 위해서이다. 이를 반복문으로 나타내면 v_θ를 0 으로 설정하고 각각의 날짜마다 다음 θ_t 를 얻고 그리고 v_θ := β*v_θ+(1-β)θ_t 로 업데이트 된다.

이렇게 **지수평균을 얻는 식의 장점**은 **아주 적은메모리를 사용**한다는 것이다. v_θ 실수 하나만을 컴퓨터 메모리에 저장하고 가장 최근에 얻은 값을 식에 기초에 덮어쓰기만 하면 되기 때문이다. 한줄의 코드만 작성하면 되서 효율적이고 지수가중평균을 계산하기 위해 하나의 실수를 저장하는 메모리만 필요하다.

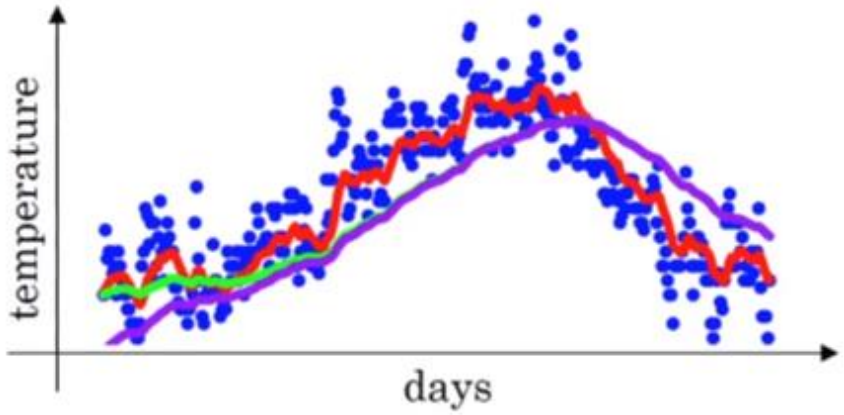
지수 가중 이동 평균의 편향보정

편향보정은 평균을 더 정확하게 계산할 수 있게 한다.

Bias correction



$$\rightarrow v_t = \beta v_{t-1} + (1-\beta)\theta_t$$



$$\rightarrow v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

작성된대로 공식을 구현하다면 **β=0.98** 일때 초록색 곡선을 얻지 못한다. 오른쪽 그래프의 보라색 곡선을 얻게 된다. 보라색 곡선이 매우 낮은 곳에서 시작함을 알 수 있다. 이를 고쳐보자.

이동평균을 구할 때 v_0 = 0 으로 초기화하고 v_1 = 0.98*v_0 + 0.02*θ_1 이다. 그러나 v_0 가 0 이기 때문에 오른쪽 식의 첫 항은 사라지게 된다. 따라서 첫째날의 온도가 화씨 40 도면 v_1 의 값은 0.02*40 인 8 이 될 것이다. 값이 훨씬 낮아져서 첫번째 날의 온도를 잘 추정할 수 없다. v_2 = 0.98*v_1 + 0.02*θ_2 가 될 것이다. v_1 값을 대입하면 v_2 = 0.98*0.02*θ_1 + 0.02*θ_2 가 된다. 이는 0.0196*θ_1 + 0.02*θ_2 이다. θ_1 과 θ_2 가 양수라고 가정한다면 v_2 를 계산한 값은 θ_1 이나 θ_2 보다 훨씬 더 작아질 것이다. 한 해의 첫 두 날짜를 추정한 값이 좋지 않은 추정이 된다.

$$\rightarrow v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$V_0 = 0$

$V_1 = 0.98 V_0 + 0.02\,\theta_1$

$V_2 = 0.98\, V_1 + 0.02\,\theta_2$

$= 0.98 \times 0.02 \times \theta_1 + 0.02\,\theta_2$

$= 0.0196\,\theta_1 + 0.02\,\theta_2$

따라서 이 추정값이 더 나은 값이 될 수 있도록 수정하는 방법이 있다. 특히 추정의 초기단계에서 더 정확하게 보정이 가능하다. v_t 를 취하는 대신에 v_t/(1-β^t)를 취한다. (t : 현재의 온도) 예를 들어 t=2 인 경우는 1-β^t = 1-(0.98)^2 = 0.0396 이다. 따라서 둘째날의 온도를 추정한 값은 v_2 를 0.0396 으로 나눈 값과 같다. v2/0.0396 = (0.0196*θ_1 + 0.02*θ_2) / 0.0396 이다. 따라서 이는 θ_1 과 θ_2 의 가중평균에 편향을 없앤 값이 된다. t 가 커질수록 β^t 는 0 에 가까워진다. 그러므로 t 가 충분히 커지면 편향보정은 그 효과가 거의 없어진다. t 가 커질때 보라색 곡선과 초록색 곡선이 겹치는 이유이다.

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

$t=2: \quad 1-\beta^t = 1-(0.98)^2 = 0.0396$

$$\frac{V_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

그러나 초기 단계의 학습에서 편향보정은 더 나은 온도의 추정값을 얻을 수 있도록 도와준다. 보라색 선에서 초록색 선으로 갈 수 있게 한다.

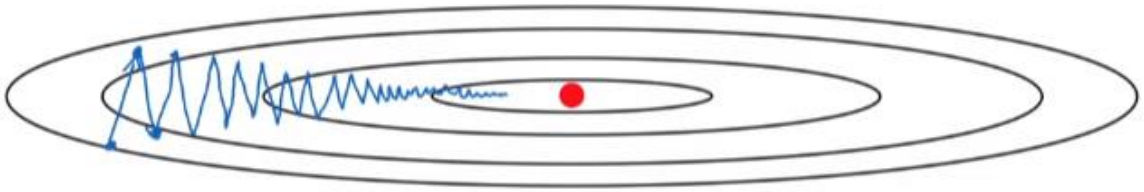
머신러닝에서 지수가중 평균을 구현하는 대부분의 경우 사람들은 편향보정을 거의 구현하지 않는다. 초기 단계를 그냥 기다리고 편향된 추정이 지나간 후부터 시작하기 때문이다. 그러나 초기 단계의 편향이 신경쓰인다면 편향보정은 초기에 더 나은 추정값을 얻는데 도움이 될 것이다.

Momentum 최적화 알고리즘

모멘텀 알고리즘 혹은 모멘텀이 있는 경사하강법은 **일반적인 경사 하강법보다 거의 항상 빠르게 작동**한다. 기본적인 아이디어는 **경사에 대한 지수가중평균을 계산**하는 것이다. 그 값으로 가중치를 업데이트한다. 어떻게 구현할 수 있을지 알아보자.

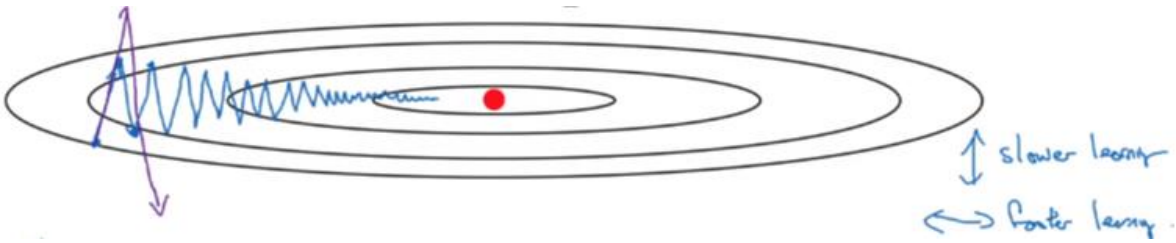
Gradient descent example

대부분의 예제에서 **비용함수를 최적화**한다고 가정하자. 등고선은 다음과 같고 빨간점은 최소값의 위치를 나타낸다. 경사하강법을 시작해서 경사하강법 or 미니배치 경사 하강법의 한 반복을 취하면 그림과 같이 향한다. 타원의 반대쪽에서 경사하강법의 한 단계를 취하면 그림과 같이 오게 된다. 이런 식으로 계속 한단계씩 나아갈 때마다 그림과 같이 나아가게 된다. 많은 단계를 취하면 최소값으로 나아가면서 천천히 진동한다.

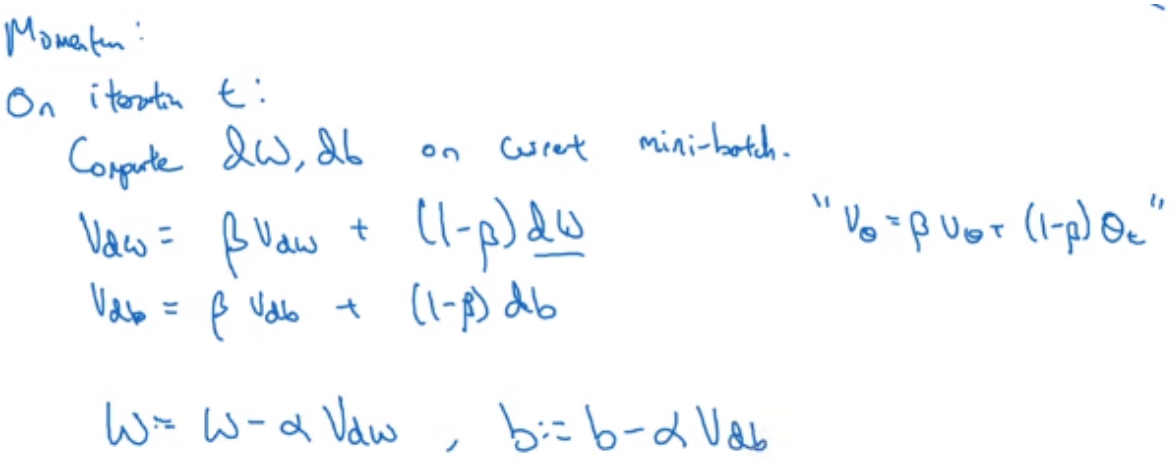


위아래로 일어나는 진동은 경사하강법의 속도를 느리게 하고 더 큰 학습률을 사용하는 것을 막는다. 왜냐하면 오버슈팅하게 되어 발산할 수도 있기 때문이다. 따라서 학습률이 너무 크지 않아야 진동이 커지는 것을 막을 수 있다.

또 다른 관점에서 살펴보면 수직축에서는 진동을 막기위해 학습이 더 느리게 일어나길 바라지만 수평축에서는 더 빠른 학습을 원한다. 최소값을 향해 왼쪽에서 오른쪽으로 이동하는 것을 처리하고 싶기 때문이다.



따라서 모멘텀을 이용한 경사하강법에서는 구현할 때에는 다음과 같이 한다. 각 반복 t 에서 현재의 미니배치에 대한 보편적인 도함수인 dw 와 db 를 계산하게 될 것이다. (배치 경사하강법을 사용하는 경우 현재의 미니배치는 전체 배치와 같다.) $V_{dw} = \beta V_{dw} + (1-\beta)dw$ 를 계산한다. (이동평균을 w 에 대한 도함수로 계산) $V_{db} = \beta V_{db} + (1-\beta)db$ 를 계산한다. 그리고 나서 w 를 사용해 가중치를 업데이트한다. $w := w - aV_{dw}$, $b := b - aV_{db}$ 가 된다.



- 알고리즘은 아래와 같습니다.
  - $V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW$
  - $w := w - \alpha V_{dW}$

Momentum 의 장점은 경사하강법의 단계를 부드럽게 만들어준다. 경사하강법은 결국에 수직방향에서는 훨씬 더 작은 진동이 있고 수평방향에서는 더 빠르게 움직인다는 것을 찾을 수 있다. 따라서 이 알고리즘은 더 직선의 길을 가거나 진동을 줄일 수 있게 한다.

이 모멘텀에서 얻을 수 있는 직관은 밥그릇 모양의 함수를 최소화하려고 하면 **도함수의 항들(dw, db)**은 아래로 내려갈 때 **가속을 제공**한다고 볼 수 있다. 그리고 **모멘텀 항들(V_dw, V_db)**은 **속도를 나타낸다**고 볼 수 있다. 따라서 작은 공이 밥그릇의 경사를 내려갈때 도함수는 여기에 가속을 부여하고 더 빠르게 내려가게 만든다. 그리고 **β는 1 보다 조금 작기 때문**에 **마찰을 제공해서 공이 제한없이 빨라지는 것을 막는다**. 따라서 경사하강법이 모든 이전 단계를 독립적으로 취하는 대신에 그릇을 내려가는 공에 가속을 주고 모멘텀을 제공할 수 있다.

Implementation details

이제 어떻게 구현할지에 대한 세부사항을 살펴보자. 여기 학습률 a 와 지수가증평균을 제어하는 β라는 두가지 하이퍼파라미터가 있다. β의 가장 일반적인 값은 0.9 이다. 지난 10 일간의 온도를 평균하는 것이다. 0.9 인경우 실제로 매우 잘 작동한다. 다양한 값을 시도하면서 하이퍼파라미터를 탐색해라.

On iteration t:

Compute dW, db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1-\beta)dW$$

$$v_{db} = \beta v_{db} + (1-\beta)db$$

$$W = W - \alpha v_{dW},\ b = b - \alpha v_{db}$$

Hyperparameters: α, β $\beta = 0.9$

편향 보정은 어떨까? V_dw/(1-β^t)이다. 많이 사용하지 않는다.

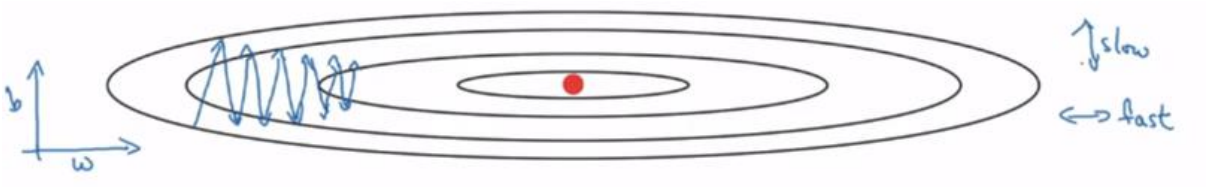
왜냐하면 **10 단계의 반복이 넘어가면 이동평균이 충분히 진행되어 편향 추정이 더이상 일어나지 않기 때문**이다. 따라서 경사하강법이나 모멘텀을 구현할 때 편향보정을 하는 사람들은 거의 없다.

모멘텀이 있는 경사하강법은 모멘텀이 없는 경사하강법보다 거의 항상 더 잘 작동한다. 그러나 학습 알고리즘을 빠르게 하기 위한 또 다른 방법이 있다. 아래에서 살펴보자.

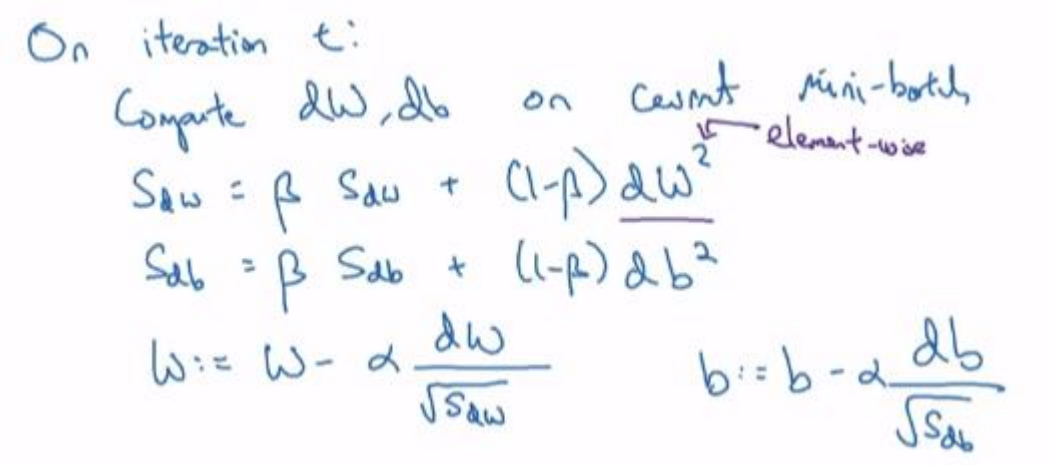
RMSProp 최적화 알고리즘

RMSProp

위에서 했던 예제를 살펴보면 경사하강법에서 수평방향으로 진행을 시도해도 수직방향으로 큰 진동이 있다는 것을 알 수 있다. 직관을 위해 수직축은 매개변수 b, 수평축은 매개변수 w 라고 하자. b 방향 또는 수직방향의 학습 속도를 낮추기 위한 것이고 그리고 수평방향의 속도를 빠르게 하기 위한 것이다.



RMSProp 알고리즘이 하는 일은 다음과 같다. 반복 t 에서 현재의 미니배치에 대한 보통의 도함수 dw 와 db 를 계산할 것이다. 지수가중평균을 유지하기 위해서 새로운 표기법인 s_dw 를 사용하자. 이 값은 β*s_dw + (1-β)*dw^2 이다. 제곱 표시는 요소별 제곱을 나타낸다. 이는 도함수의 제곱을 지수가중평균하는 것이다. 그리고 s_db 역시 β*s_db + (1-β)*db^2 와 같다. 다음으로 매개변수를 다음과 같이 업데이트 한다. w 는 w 에서 학습률 a*dw 를 s_dw 의 제곱근으로 나눠준 값을 뺀 것이다. b 는 b-a*db 만을 하는 대신에 s_db 의 제곱근으로 나눠준 값을 뺀다.



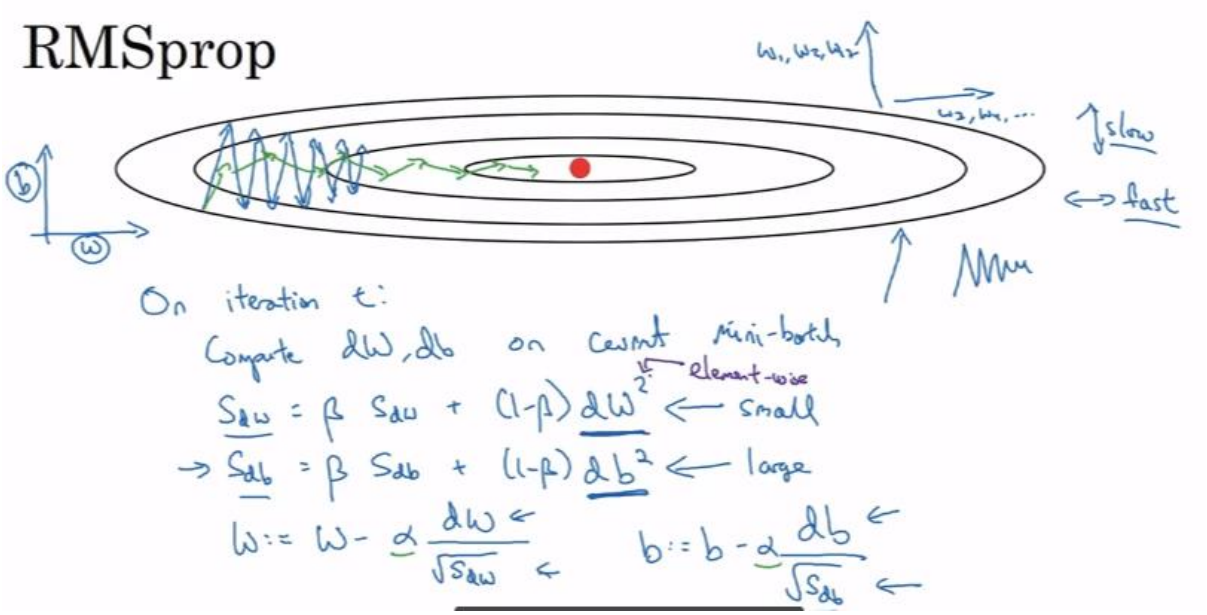
- 알고리즘은 아래와 같습니다.
  - $S_{dW} = \beta_2 S_{dW} + (1-\beta_2) dW^2$
  - 업데이트: $w := w - \alpha \frac{dW}{\sqrt{S_{dW} + \epsilon}}$
  - dW^2 은 요소별 제곱을 뜻합니다.

이것이 어떻게 작동하는걸까?

수평 방향(w 방향)에서는 **학습률이 꽤 빠르게** 가길 원하는 반면에 **수직 방향(b 방향)**에서는 **느리게 혹은 수직방향의 진동을 줄이길** 원한다. 따라서 s_dw 와 s_db 에서 우리가 원하는 것은 s_dw 가 상대적으로 작고 s_db 가 상대적으로 큰 것이다. (s_dw 로 상대적으로 작게 나누고 s_db 로 상대적으로 크게 나눈다는 의미) 수직방향에서의 업데이트를 줄이기 위함이다. 실제로 수직방향에서의 도함수가 수평방향의 것보다 훨씬 크다. b 방향에서의 경사가 매우 크다. 도함수 db 는 매우 크고 dw 는 상대적으로 작다. (수직방향 b 에서 기울기가 더 가파르기 때문) 수평방향 w 에서의 기울기보다 가파르다. 따라서 db^2 은 상대적으로 크고 dw 가 작기 때문에 dw^2 은 상대적으로 더 작다. **다음에 오는 효과는 더 큰 숫자로 나눠서 수직방향에서 업데이트하기 때문에 진동을 줄이는데 도움을 준다.** 반면 **수평방향에서는 작은 숫자로 나눠서 업데이트하기 때문에** RMSProp 을 사용한 업데이트는 다음과 같다. - 수직방향에서의 업데이트는 감소하지만

수평방향은 계속 나아가게 한다. 이 효과는 **큰 학습률을 사용해 빠르게 학습하고 수직방향으로 발산하지 않는다.**

명확히는 수직과 수평방향을 b 와 w 로 나타냈는데 실제로는 매우 매개변수의 고차원 공간에 있기 때문에 진동을 줄이려는 수직차원은 w1, w2, ... w17 의 매개변수 집합이고 수평방향의 차원은 w3, w4.. 처럼 나타날 것이다. 따라서 w 와 b 의 분리는 표현을 위한 것이고 실제로 dw 와 db 는 매우 고차원의 매개변후 벡터이다.



RMSProp 의 장점은 미분값이 큰 곳에서는 업데이트 시 큰 값으로 나눠주기 때문에 기존 학습률 보다 작은 값으로 업데이트 됩니다. 따라서 진동을 줄이는데 도움이 됩니다. 반면 미분값이 작은 곳에서는 업데이트시 작은 값으로 나눠주기 때문에 기존 학습률 보다 큰 값으로 업데이트 됩니다. 이는 더 빠르게 수렴하는 효과를 불러옵니다.

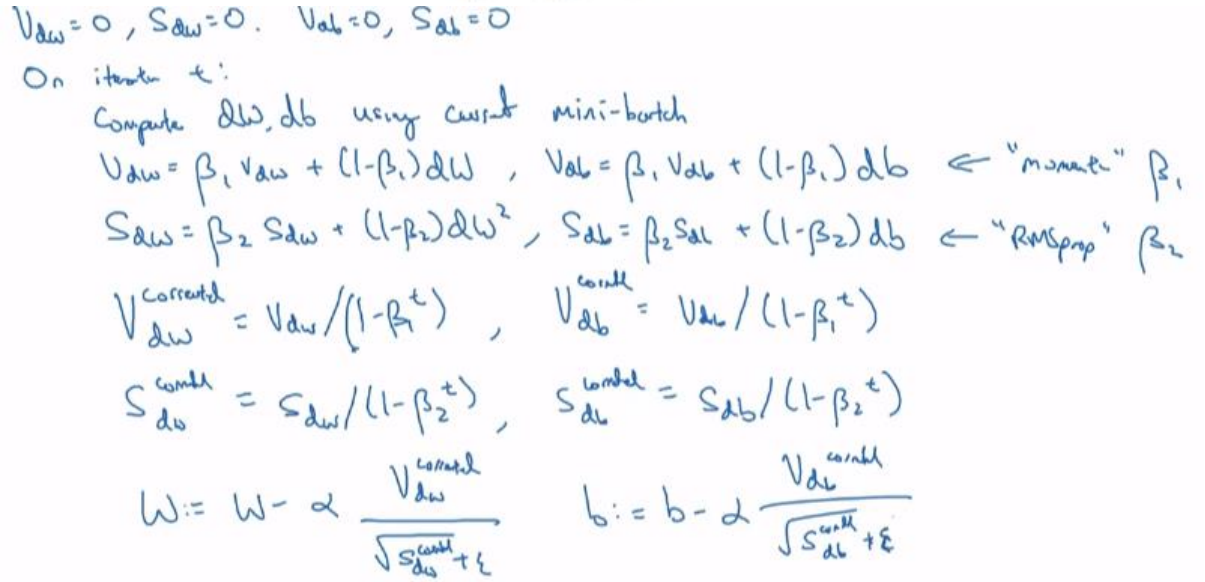
RMSProp 알고리즘은 학습 알고리즘의 속도를 올리는 방법이다. RMSProp 와 모멘텀을 함께 사용하면 더나은 최적화 알고리즘을 얻을 수 있다

Adam 최적화 알고리즘

Adam 최적화 알고리즘은 RMSprop 과 모멘텀을 합친 알고리즘이다. 어떻게 동작하는지 살펴보자.

Adam optimization algorithm

- 알고리즘은 아래와 같습니다.
  - $V_{dW} = 0, S_{dW} = 0$ 로 초기화 시킵니다.
  - Momentum 항: $V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW$
  - RMSProp 항: $S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2$
  - Bias correction: $V_{dW}^{correct} = \frac{V_{dW}}{1 - \beta_1^t}, S_{dW}^{correct} = \frac{S_{dW}}{1 - \beta_2^t}$
  - 업데이트: $w := w - \alpha \frac{V_{dW}^{correct}}{\sqrt{S_{dW}^{correct}} + \epsilon}$



전형적인 Adam 구현에서는 편향보정을 한다. V_dw^correctedf 는 편향보정을 의미한다. 그리고 최종적으로 업데이트를 실행한다. 위 알고리즘은 모멘텀이 있는 경사하강법의 효과와 RMSprop 이 있는 경사하강법의 효과를 합친 결과가 나온다. 이는 매우 넓은 범위의 아키텍처를 가진 서로 다른 신경망에서 잘 작동한다는 것이 증명된 일반적으로 많이 쓰이는 학습알고리즘이다.

따라서 이 알고리즘은 **많은 하이퍼파라미터**가 있다.

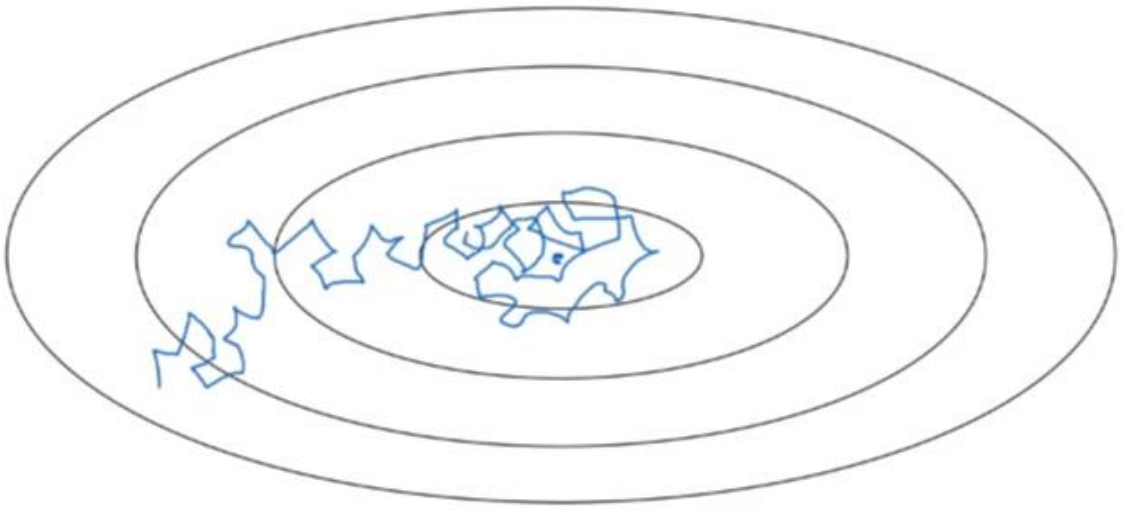
- 학습률 하이퍼파라미터 **a** : 매우 중요하고 보정될 필요가 있으므로 **다양한 값을 시도**해서 잘 맞는 것을 찾아야 한다.
- **β_1** : 기본적인 값으로 **0.9** 를 보통 선택한다. (dw 의 이동평균, 가중평균/ 모멘텀에 관한 항)
- **β_2** : Adam 논문에서 저자가 추천하는 값이 **0.999** 이다. (dw^2 와 db^2 의 이동가중평균을 계산한 것)
- **ε** : **10^(-8)**의 값을 추천 (이 값을 설정하지 않더라도 전체 성능에 영향은 없음)

Adam 최적화 알고리즘을 이용하면 신경망을 더 빠르게 훈련시킬 수 있을 것이다.

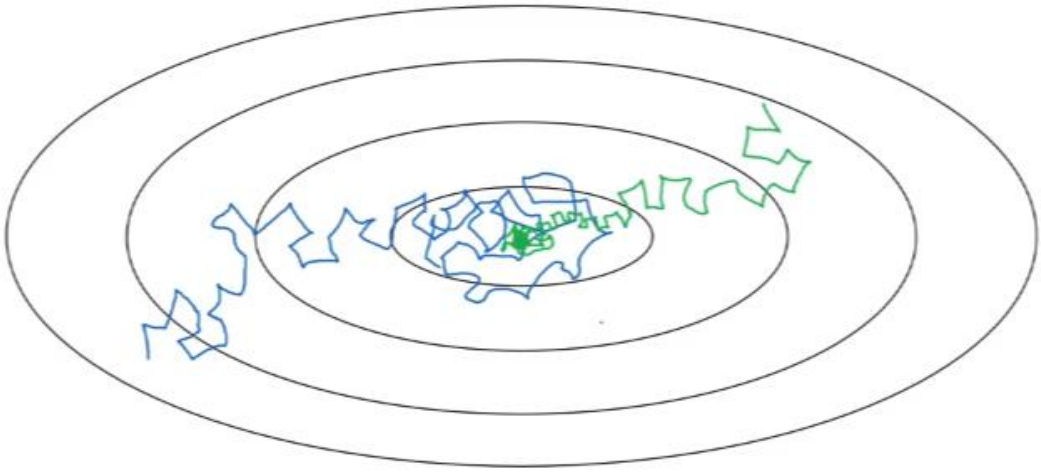
학습률 감쇠

Learning rate decay

왜 학습률 감쇠가 필요한지 예시를 하나 보자. **상당히 작은 미니배치(64, 128)에 대해 미니배치 경사 하강법을 구현**한다고 가정하자. 이 경우, 단계를 거치면서 **약간의 노이즈가 있지만 최소값으로 향하는 경향**을 보일 것이다. 그러나 정확하게는 수렴하지 않고 주변을 돌아다니게 된다. 왜냐하면 어떤 고정된 값인 a 를 사용했고 서로 다른 미니배치에 노이즈가 있기 때문이다.



그러나 **천천히 학습률 a 를 줄이면** a 가 여전히 **큰 초기 단계에서는 상대적으로 빠른 학습**이 가능하다. a 가 작아지면 단계마다 진행정도가 작아지고 **최소값의 밀집된 영역에서 진동**하게 될 것이다. 훈련이 계속되더라도 최소값 주변에 배회하는 대신에 말이다. 따라서 a 를 천천히 줄이는 것의 의미는 학습 초기 단계에서는 훨씬 큰 스텝으로 진행하고 학습을 수행할 수록 학습률이 느려져 작은 스텝으로 진행하는 것이다.



따라서 여기 학습률 감쇠를 구현하는 방법이다.

- 1 epoch = 전체 데이터를 1번 훑고 지나가는 횟수 입니다.
- $\alpha = \frac{1}{1 + decay\ rate \times epoch\ num}\alpha_0$
- $\alpha = 0.95^{epoch\ num}\alpha_0$ (exponential decay 라고 부릅니다.)
- $\alpha = \frac{k}{\sqrt{epoch\ num}}\alpha_0$
- $\alpha = \frac{k}{\sqrt{batch\ num}}\alpha_0$
- step 별로 α 다르게 설정

하나의 에포크는 데이터를 지나는 하나의 패스다. 훈련세트를 서로 다른 미니배치로 나눠서 훈련세트를 지나는 첫번째 패스를 첫번째 에포크라고 부른다. 두번째 지나는 것을 두번째 에포크라고 부른다. 에포크수에 대한 함수에서 학습률은 점차적으로 감소한다. 학습률 감쇠를 사용하고 싶다면 하이퍼파라미터 a_0 와 감쇠율에 대해서 다양한 값을 시도하고 잘 작동하는 값을 찾으면 된다.

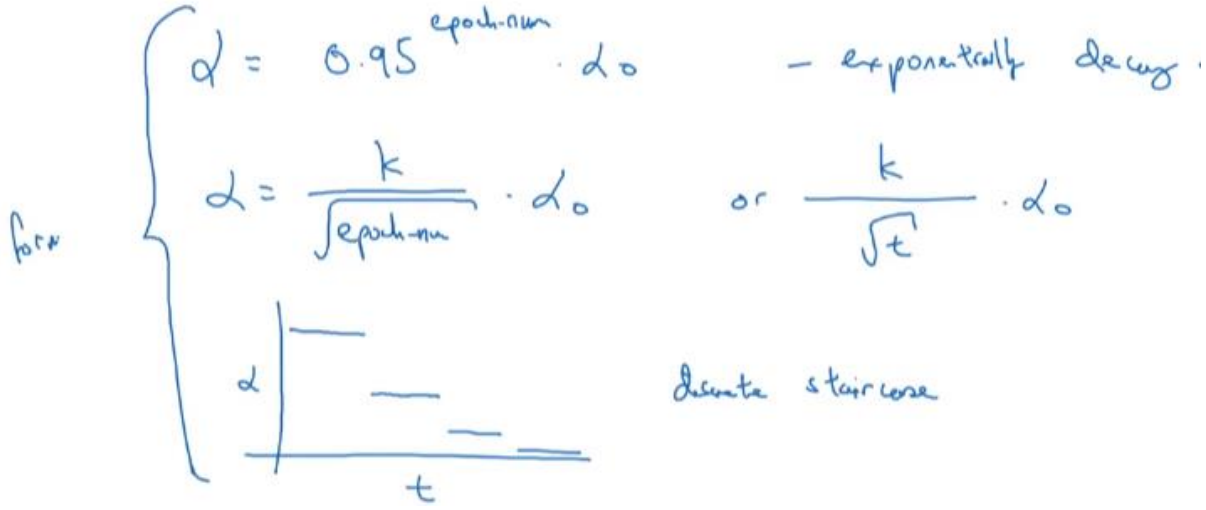
Other learning rate decay methods

학습률 감쇠에 대한 이 식 말고 사람들이 사용하는 또 다른 방법들이 있다. 예를 들면 **지수적 감쇠**라고 불리는 것은 a 가 1 보다 작은 값을 갖는다.

a = 0.95^epoch_num*a_0 -> 이것은 기하급수적으로 빠르게 학습률을 감소시킨다.

a = k(상수)/epoch_num 의 제곱근*a_0 또는 **k(상수)/미니배치의 개수 t 의 제곱근*a_0**

또 어떤 사람들은 **이산적 단계로 감소하는 학습률**을 사용하기도 한다. 어떤 단계에서는 어떤 학습률 값을 가지고 그 뒤에는 학습률이 반으로 줄어들고 일정 시간이 지날 때마다 계속 반씩 줄어드는 모습니다.



지금까지 시간에 따라 학습률이 어떻게 바뀌는지를 통제하는 여러가지 식들을 살펴봤다. 사람들이 사용하는 또다른 방법은 **직접 조작하는 감쇠**이다. 한 번에 하나의 모델을 훈련하는데 몇시간 혹은 며칠이 걸린다면 어떤 사람들은 훈련을 거치면서 모델을 정리해 나갈 것이다. **학습률이 느려지고 있는 것처럼 느껴서 데이터의 크기를 줄이는 것**이다. 이런 식으로 a 의 값을 시간이나 날마다 직접 보정하는 것은 **훈련이 작은 수의 모델로만 이루어진 경우에 가능**하다.