Copyright Notice

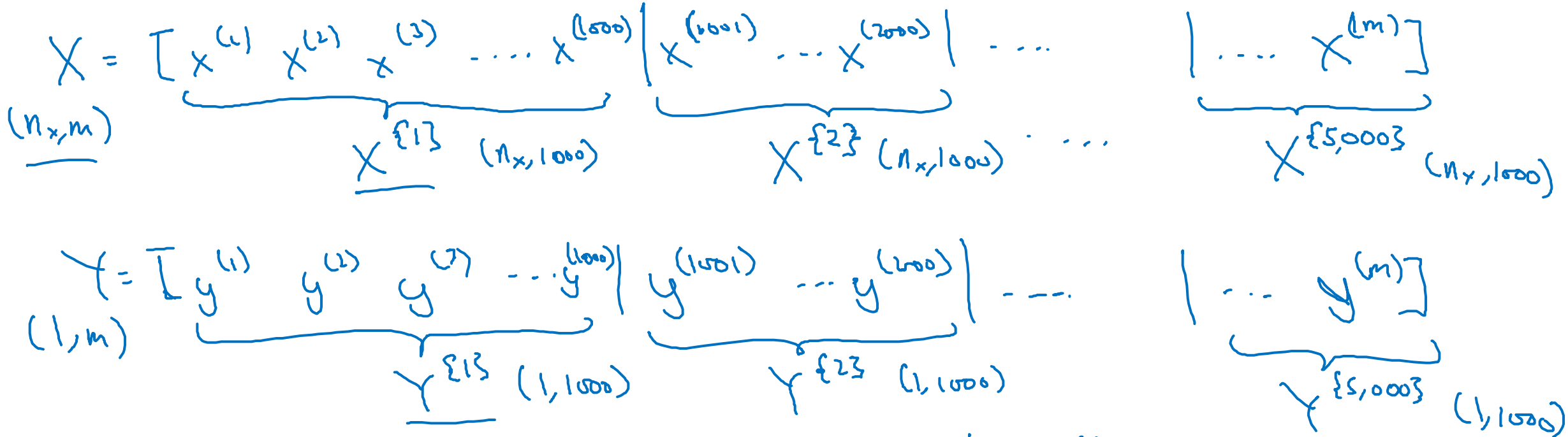
deeplearning.ai

Optimization Algorithms

Mini-batch gradient descent

Batch vs. mini-batch gradient descent

Vectorization allows you to efficiently compute on m examples.



What if $m = 5{,}000{,}000$?

5,000 mini-batches of 1,000 each

Mini-batch t: $X^{\{t\}}, Y^{\{t\}}$.

$x^{(i)}$

$z^{[l]}$

$X^{\{t\}}, Y^{\{t\}}$.

Andrew Ng

Mini-batch gradient descent

repeat {

for $t = 1, \ldots, 5000$ { : 전체 iteration을 돌릴 외부의 for loop 가 필요하다.

← 1 step of gradient descent using $X^{\{t\}}, Y^{\{t\}}$ (as if $m = 1000$)

X, Y

Forward prop on $X^{\{t\}}$.

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$\vdots$$

$$A^{[L]} = g^{[L]}(Z^{[L]})$$

} Vectorized implementation (1000 examples)

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l} \|W^{[l]}\|_F^2$.

($\hat{y}^{(i)}, y^{(i)}$ from $X^{\{t\}}, Y^{\{t\}}$.)

Backprop to compute gradients wrt $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)

$$W^{[l]} := W^{[l]} - \alpha dW^{[l]}, \quad b^{[l]} := b^{[l]} - \alpha db^{[l]}$$

}

}

"1 epoch"
└ pass through training set.

Andrew Ng

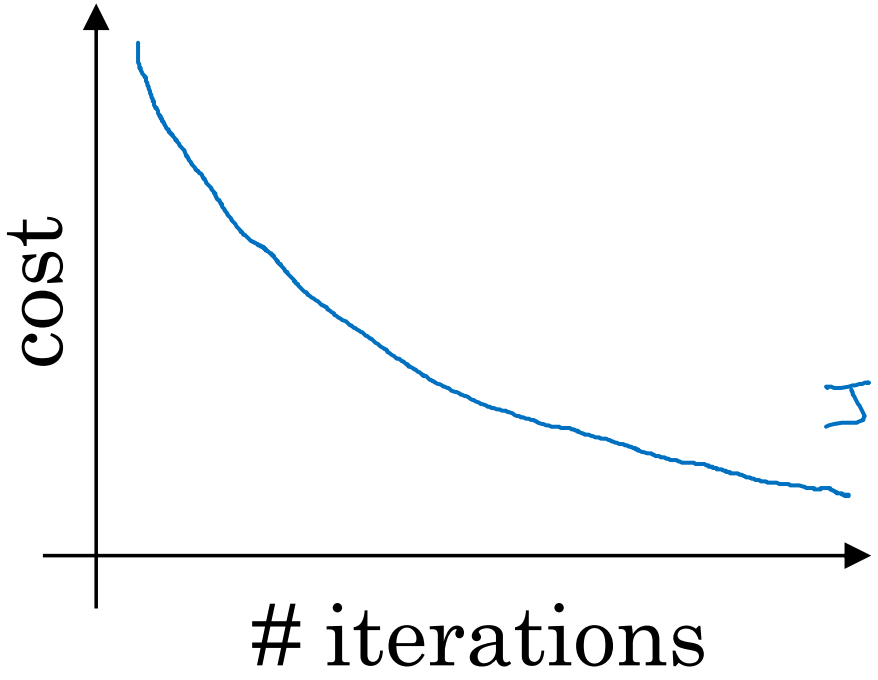
deeplearning.ai

Optimization Algorithms

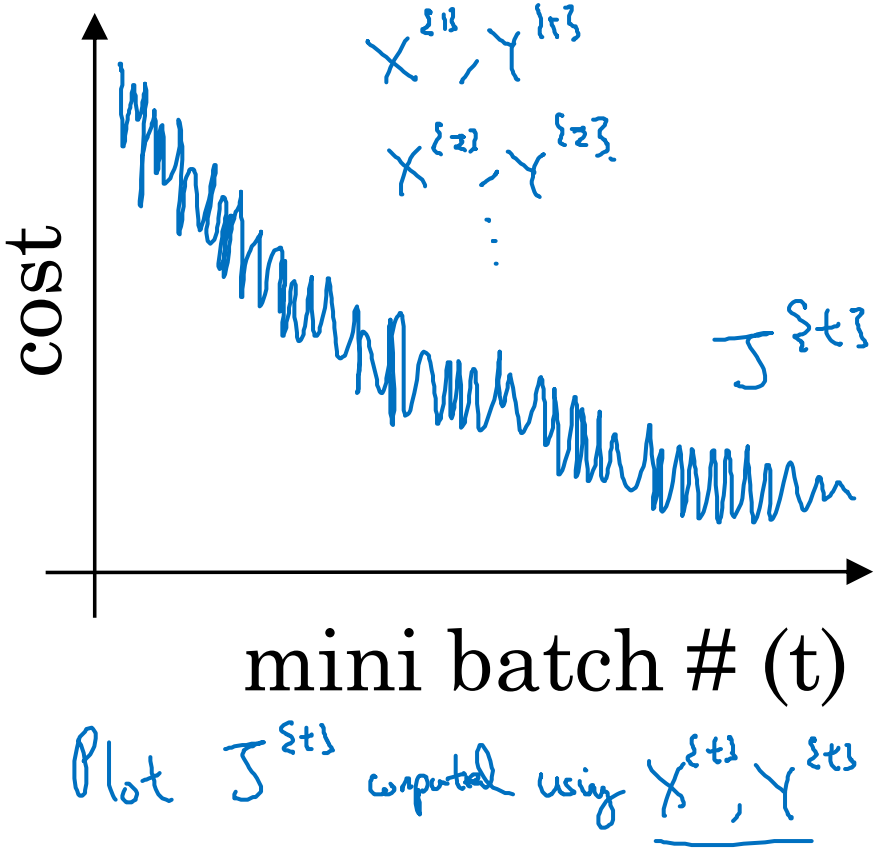
Understanding mini-batch gradient descent

Training with mini batch gradient descent

Batch gradient descent



Mini-batch gradient descent

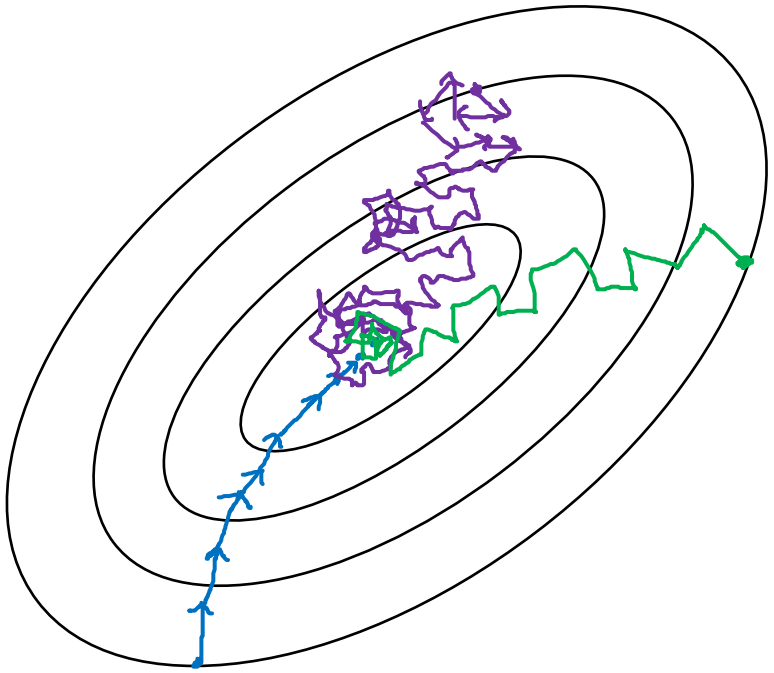


Choosing your mini-batch size

→ If mini-batch size = m : Batch gradient descent. $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$.

→ If mini-batch size = 1 : Stochastic gradient descent. Every example is it own mini-batch. $(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)})$ ... $(x^{(2)}, y^{(1)})$

In practice: Somewhere in-between 1 and m



Stochastic gradient descent

{

Lose speedup from vectorization

In-between (mini-batch size not too big/small)

Fastest learning.

- Vectorization. (n large)
- Make progress without processing entire training set.

Batch gradient descent (mini-batch size = m)

Too long per iteration

Andrew Ng

Choosing your mini-batch size

If small train set: Use batch gradient descent.
($m \leq 2000$)

Typical mini-batch sizes:

→ 64, 128, 256, 512 (1024)
2^6, 2^7, 2^8, 2^9 (2^{10})

Make sure mini-batch fit in CPU/GPU memory.

$X^{\{t\}}, Y^{\{t\}}$

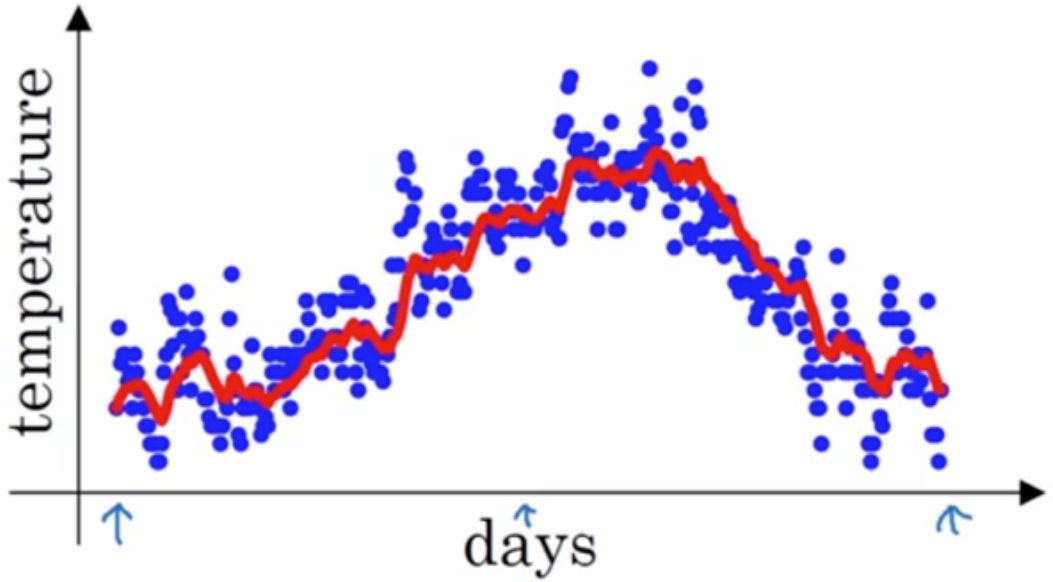
deeplearning.ai

Optimization Algorithms

Exponentially weighted averages

Temperature in London

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



$v_0 = 0$
$v_1 = 0.9\, v_0 + 0.1\, \theta_1$
$v_2 = 0.9\, v_1 + 0.1\, \theta_2$
$v_3 = 0.9\, v_2 + 0.1\, \theta_3$
⋮
$v_t = 0.9\, v_{t-1} + 0.1\, \theta_t$

Andrew Ng

Exponentially weighted averages

Moving

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

$\beta = 0.9$: $\approx$ 10 days' temperature. (= 0.9일 동안의 평균기온 변화)

$\beta = 0.98$: $\approx$ 50 days

$\beta = 0.5$: $\approx$ 2 days

V_t as approximately

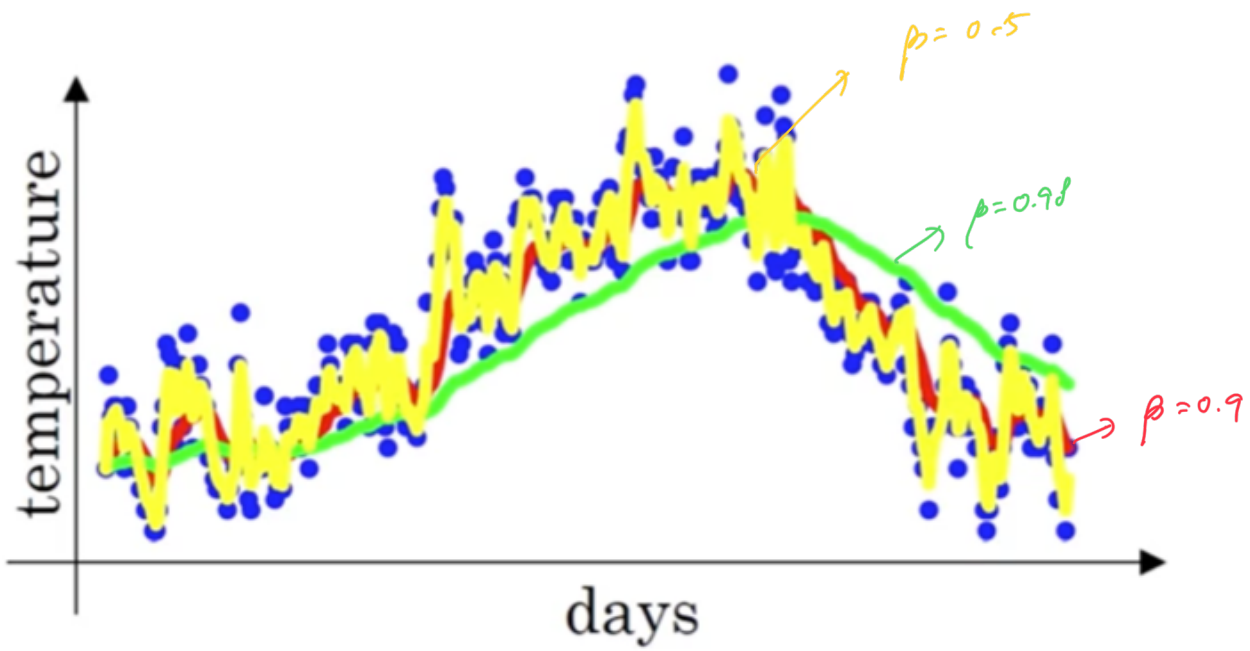
Average over

$\rightarrow \approx \frac{1}{1-\beta}$ days'

temperature.

(= V_t는 대략적으로 $\frac{1}{1-\beta}$ days의 평균기온이 된다.)

$\frac{1}{1-0.98} = 50$



deeplearning.ai

Optimization Algorithms

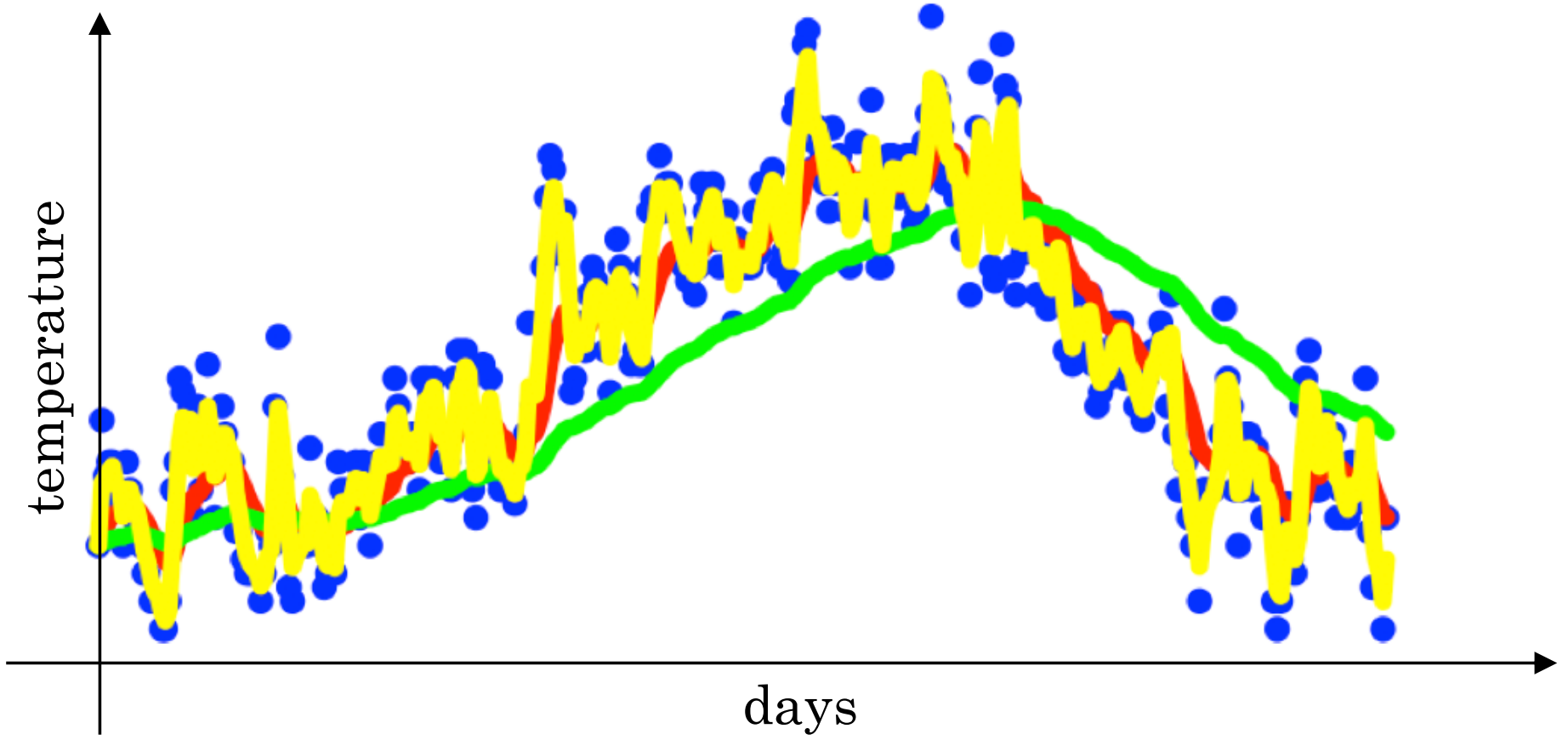
Understanding exponentially weighted averages

Exponentially weighted averages

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

$\beta = 0.9$ 0.98 0.5

지수 가중 평균 : 시간의 흐름에 따른 온도와 같은 데이터의 변화이 지수로 강조하고 싶은 것.



Andrew Ng

Exponentially weighted averages

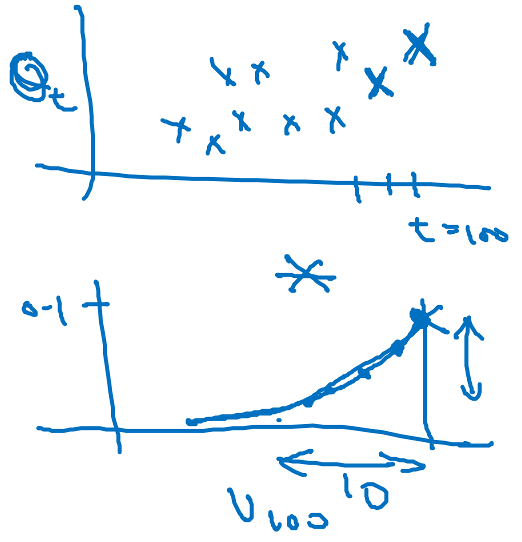
$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$$v_{100} = 0.9 v_{99} + 0.1\theta_{100}$$

$$v_{99} = 0.9 v_{98} + 0.1\theta_{99}$$

$$v_{98} = 0.9 v_{97} + 0.1\theta_{98}$$

…



$$\approx \frac{1}{1-\beta} \qquad \varepsilon = 1-\beta$$

$$0.1\theta_{98} + 0.9 v_{97}$$

$$v_{100} = 0.1\theta_{100} + 0.9\left(0.1\theta_{99} + 0.9 v_{98}\right)$$

$$= 0.1\theta_{100} + 0.1 \times 0.9 \cdot \theta_{99} + 0.1(0.9)^2\theta_{98} + 0.1(0.9)^3\theta_{97} + 0.1(0.9)^4\theta_{96} + \dots$$

$$0.9^{10} \approx 0.35 \approx \frac{1}{e} \qquad (1-\varepsilon)^{1/\varepsilon} = \frac{1}{e} \qquad 0.98\,?$$

$$\varepsilon = 0.02 \rightarrow 0.98^{50} \approx \frac{1}{e}$$

Andrew Ng

Implementing exponentially weighted averages

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

$$V_\theta := 0$$
$$V_\theta := \beta V + (1-\beta)\,\theta_1$$
$$V_\theta := \beta V + (1-\beta)\,\theta_2$$
$$\vdots$$

$V_\theta = 0$

Repeat {

  Get next θ_t

  $V_\theta := \beta V_\theta + (1-\beta)\,\theta_t$

}

장점: 기본적으로 단 한 줄의 코드로 단 하나의 값만 메모리에 저장하면 되기 때문에 아주 적은 양의 메모리가 사용되고, 효율성이 높다.

But, 위 방식은 정확한 평균치를 얻는 방식은 아니다. 정확한 평균은 최근 10일이나 50일의 기온을 더하고 10이나 50으로 나누면 더 좋은 추정치를 얻을 수 있지만, 이 방법은 그동안의 기온을 보관하고 더해야해서 더 많은 메모리가 필요하고 구현도 더 복잡하고 연산의 부담도 커지게 되는 단점이 있다.

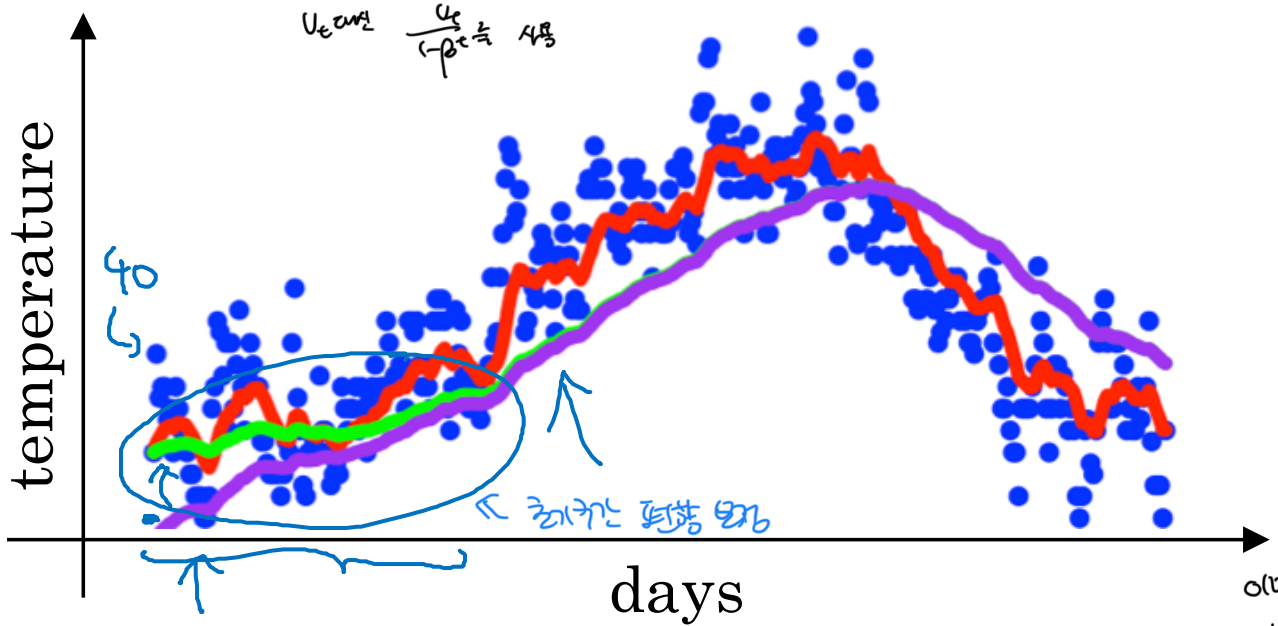
Andrew Ng

deeplearning.ai

Optimization Algorithms

Bias correction in exponentially weighted average

Bias correction



$\beta = 0.98$

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

$v_0 = 0$

$v_1 = 0.98 v_0 + 0.02\theta_1$

$v_2 = 0.98 v_1 + 0.02\theta_2$

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

$= 0.0196\theta_1 + 0.02\theta_2$

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

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

$\frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$

Andrew Ng

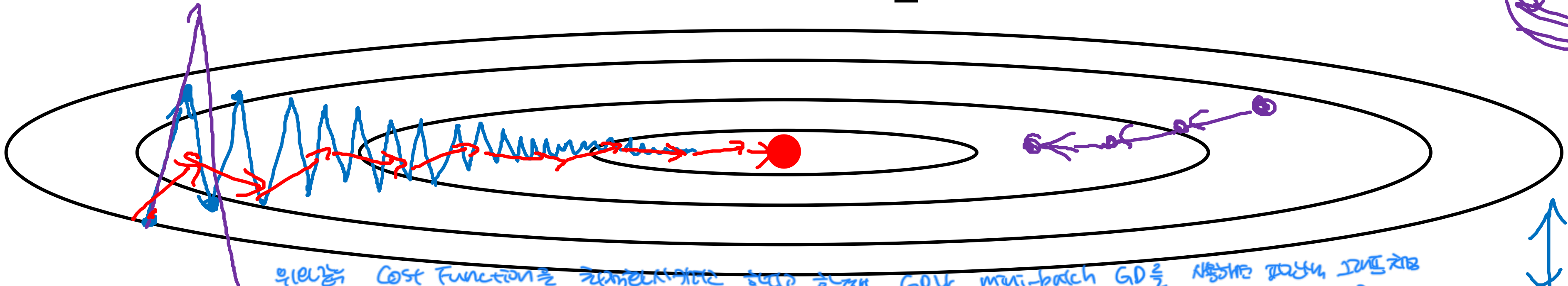
deeplearning.ai

Optimization Algorithms

Gradient descent with momentum

이 알고리즘 앞에서 Gradient descent 보다 더 빠른 것 같아서 등장했다.
이 알고리즘의 기본 아이디어는 Gradient descent의 Exponentially moving Average를 구한 것.
그 값을 이용해 파라미터 weight을 업데이트하는 것이다.

Gradient descent example



우리가 Cost Function을 최소화시키려고 한다고 할 때, GD나 mini-batch GD를 사용하면 파란색 그래프처럼 빨간 점의 최소값을 향해 앞뒤로 진동하며 천천히 접근하게 된다. 이렇게 위아래로 진동하는 현상이 GD를 느리게 한다. 만약 훨씬 더 큰 Learning Rate를 사용하는 경우에는 발산해 버릴 수도 있다. 이렇게 진동하는 폭을 감소시키기 위해서 우리는 세로축에서는 slower learning, 가로축에서는 faster learning을 원하는 것이다. 이는 GD with momentum을 통해 감소시킬 수 있다. ([illegible])

Momentum:

On iteration t:

Compute dW, db on current mini-batch.

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

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

friction → β, velocity → V_{db}, acceleration → dW, db

$$W := W - \alpha V_{dW}, \quad b := b - \alpha V_{db}$$

(알고리즘에서 dW와 dB는 가속도 역할을 하고 V_{dw}, V_{db}는 속도 역할을 한다.)

"$V_\theta = \beta V_\theta + (1-\beta)\theta_t$"

(고급방법) → 현재의 mini-batch에서 dW와 dB를 구한다. 그 다음에는 dW와 dB에 대해 EWA를 구하고, 이렇게 구한 EWA를 가지고 W와 b 파라미터를 업데이트한다. 이 과정을 통해서 Gradient Descent를 더욱 smooth하게 하려면 된다. 결과적으로 세로축의 변동 평균은 거의 0이 되고, 가로축의 변동 평균이 매우 큰 값이 되어서, 위 그림과 같이 빨간색 그래프처럼 최소값을 향해 접근

Andrew Ng

Implementation details

$V_{dW} = 0$, $V_{db} = 0$

On iteration t:

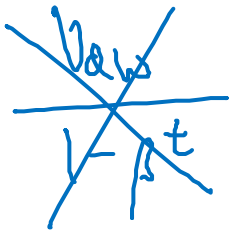
Compute dW, db on the current mini-batch

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

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

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

$V_{dW} = \beta V_{dW} + dW$ ←



Hyperparameters: α, β

$\beta = 0.9$

average over last ≈10 gradients

Andrew Ng

deeplearning.ai

Optimization Algorithms

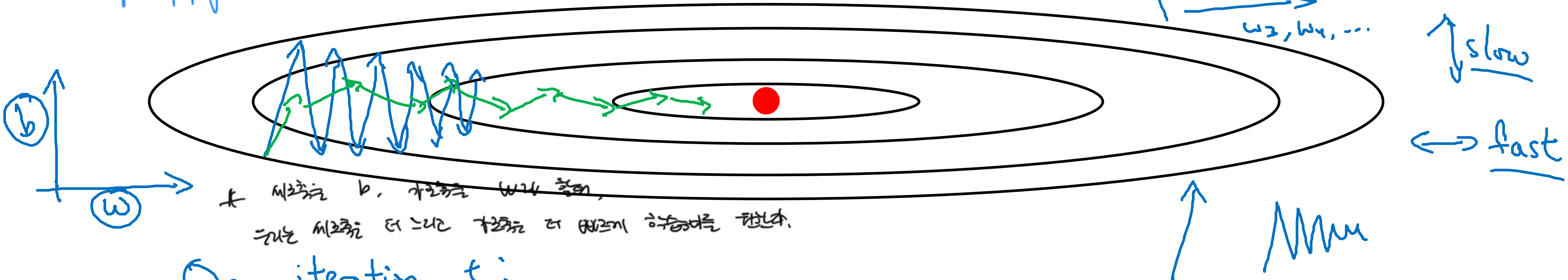
RMSprop

momentum 이용해 GD를 진행할 때보다 훨씬 빠르게 학습시킬 수 있다.
이제 또 다른 학습시킬 수 있는 learning rate 식을 사용할 수 있다.
둘을 합쳐서 속도를 증가시킬 수 있다.

RMSprop

: 이 방법 사용해 GD 속도 증가시킬 수 있다.

Root Mean Square propagation



* 세로축은 b, 가로축은 w라 할때, 우리는 세로축은 더 느리고 가로축은 더 빠르게 학습하기를 원한다.

On iteration t:

Compute dW, db on current mini-batch

(gradient)^{2}의 지수가중이동평균

$$S_{dW} = \beta_2 S_{dW} + (1-\beta_2)\, dW^2 \leftarrow \text{small}$$

$$S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2 \leftarrow \text{large}$$

element-wise

EMA의 제곱평균을 구해.

변화량이 더 클수록 학습률이 작아져서 진동하는 문제를 해결하기 위해 학습률 크기를 비율로 조정할 수 있도록 제안된 방법이다.

최근 경로의 곡면 변화량을 측정하기 위해 지수가중이동평균을 사용한다.

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}} + \varepsilon}$$

$$b := b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$$

$\sqrt{S_{dW}}$가 0에 가까워지면, 이 값이 나누어지면서 폭발적인 값을 가지게 된다. 이 값을 안정화하기 위해서 실제로 RMSprop을 구현할 때, 분모에 아주 작은 입실론(ε) 값을 더하게 된다.

$$\varepsilon = 10^{-8}$$

적응적 학습률 (최근 곡면의 변화량에 반비례)

가로방향 변화는 증가시키고 세로방향 변화는 늦추고 싶기 때문에 S_{dW}는 비교적 작은 값이 되어야 하고, S_{db}는 비교적 큰 값이 되어야 한다. 실제 미분값을 보면, 가로방향보다 세로방향으로 더 큰 것을 볼 수 있고, 기울기는 b 방향으로 더 크게 된다. 이것은 가로 방향의 변동폭은 유지하고, 세로 방향의 변동폭은 커지도록 되어지게 되고, 결국 그래프와 같은 모양으로 업데이트가 진행된다.

deeplearning.ai

Optimization Algorithms

Adam optimization algorithm

Adam optimization algorithm

: Momentum과 RMS prop 이 합쳐진 형태
+ 추가적으로 Bias Correction도 좀 더 진행함
진동하며 속도에 안정성도 주고, 최근 경로의 가중치 변화량이 더 큰 가중치, 학습률도 갖는 알고리즘
※ 일부는 학습과정에 좋아 거의 많이 사용함

$V_{dw} = 0$, $S_{dw} = 0$. $V_{db} = 0$, $S_{db} = 0$

On iteration t:

Compute dW, db using current mini-batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dW$, $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$ ← "momentum" β_1

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dW^2$, $S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$ ← "RMSprop" β_2

yhat = np.array([.9, 0.2, 0.1, .4, .9])

bias correction

$$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t), \quad V_{db}^{corrected} = V_{db} / (1-\beta_1^t)$$

$$S_{dw}^{corrected} = S_{dw} / (1-\beta_2^t), \quad S_{db}^{corrected} = S_{db} / (1-\beta_2^t)$$

$$W := W - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon} \qquad b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

Andrew Ng

Hyperparameters choice:

→ α : needs to be tune : 매우 중요하고 변경될 필요가 있으므로 다양한 값을 시도해서 잘 맞는 값을 찾아야 한다.

→ β_1 : 0.9 → (dw) : 모멘텀에 관한 항

→ β_2 : 0.999 → (dw^2)

→ ε : 10^{-8} : 분모가 0이 되는 것을 막기 위해 더해지는 상수. 하지만, 이 값을 설정하지 않아도 전체 성능에는 영향이 없다.

Adam : Adaptive moment estimation

* Adam 알고리즘을 사용할 때는 보통 기본값을 많이 사용해 β_1, β_2, ε 값은 잘 변경하지 않는다. α 값만 여러가지 값들을 시도해보고 어떤 값이 잘 동작하는지 확인해보면 된다.

Adam Coates

Andrew Ng

deeplearning.ai

Optimization Algorithms

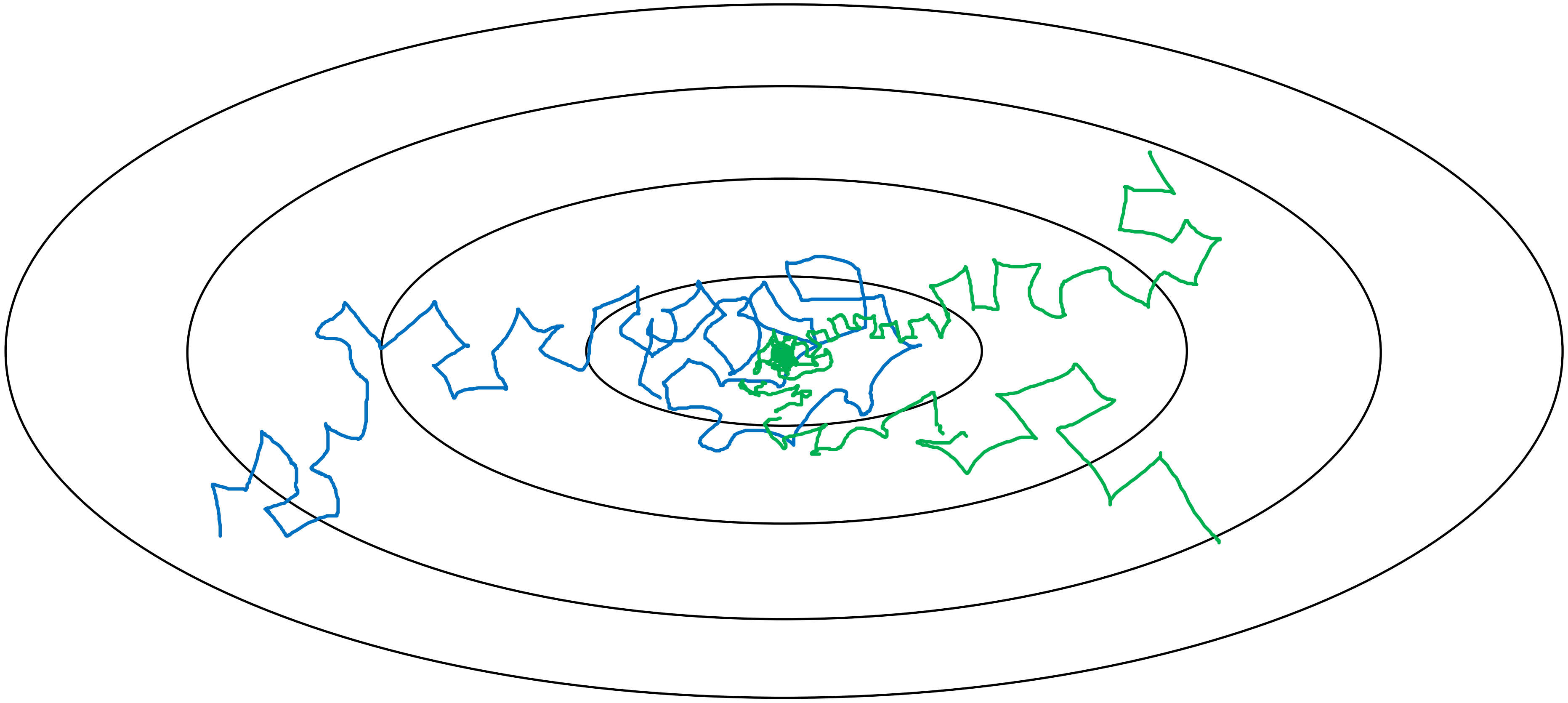
Learning rate decay

Learning rate decay

: 기존의 Learning Rate가 높은 경우 loss 값을 빠르게 내릴 수는 있지만, 최적의 학습을 벗어나게 만들고, 낮은 경우 최적의 학습을 할 수 있지만 그 단계까지 너무 오랜 시간이 걸린다.

따라서, 처음 시작시 Learning Rate 값을 크게 준 후 일정 epoch 마다 값을 감소시켜서 최적의 학습까지 더 빠르게 도달할 수 있게 하는 방법.

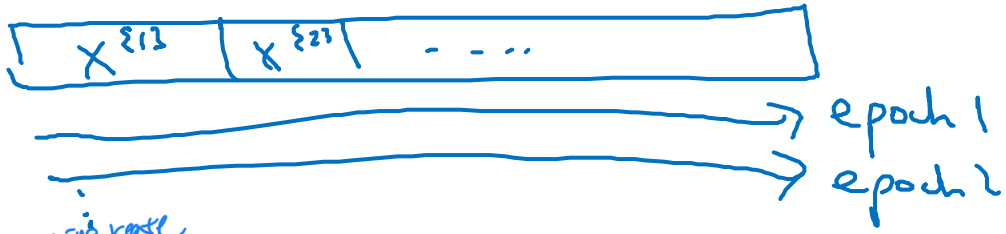
Slowly reduce α



Andrew Ng

Learning rate decay

1 epoch = 1 pass through data.



$$\alpha = \frac{1}{1 + \text{decay\_rate} * \text{epoch\_num}} \alpha_0$$

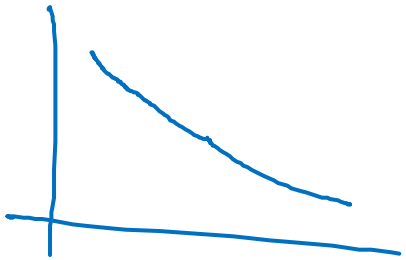
α_0: initial learning rate

epoch_num: over mini-batch computation

$\alpha_0 = 0.2$

decay_rate = 1

Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04
⋮	⋮



Andrew Ng

Other learning rate decay methods

formula

$$\alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0 \quad \text{— exponentially decay.}$$

$$\alpha = \frac{k}{\sqrt{\text{epoch-num}}} \cdot \alpha_0 \quad \text{or} \quad \frac{k}{\sqrt{t}} \cdot \alpha_0$$

discrete staircase

* 보통 learning rate decay는 나중에 시도해볼만한 것이고, 보통 α를 하나의 값으로 정해서 잘 튜닝하는 것이 중요하고 있고, 나중에 필요할 때 learning rate decay를 사용하는 것을 권장한다.

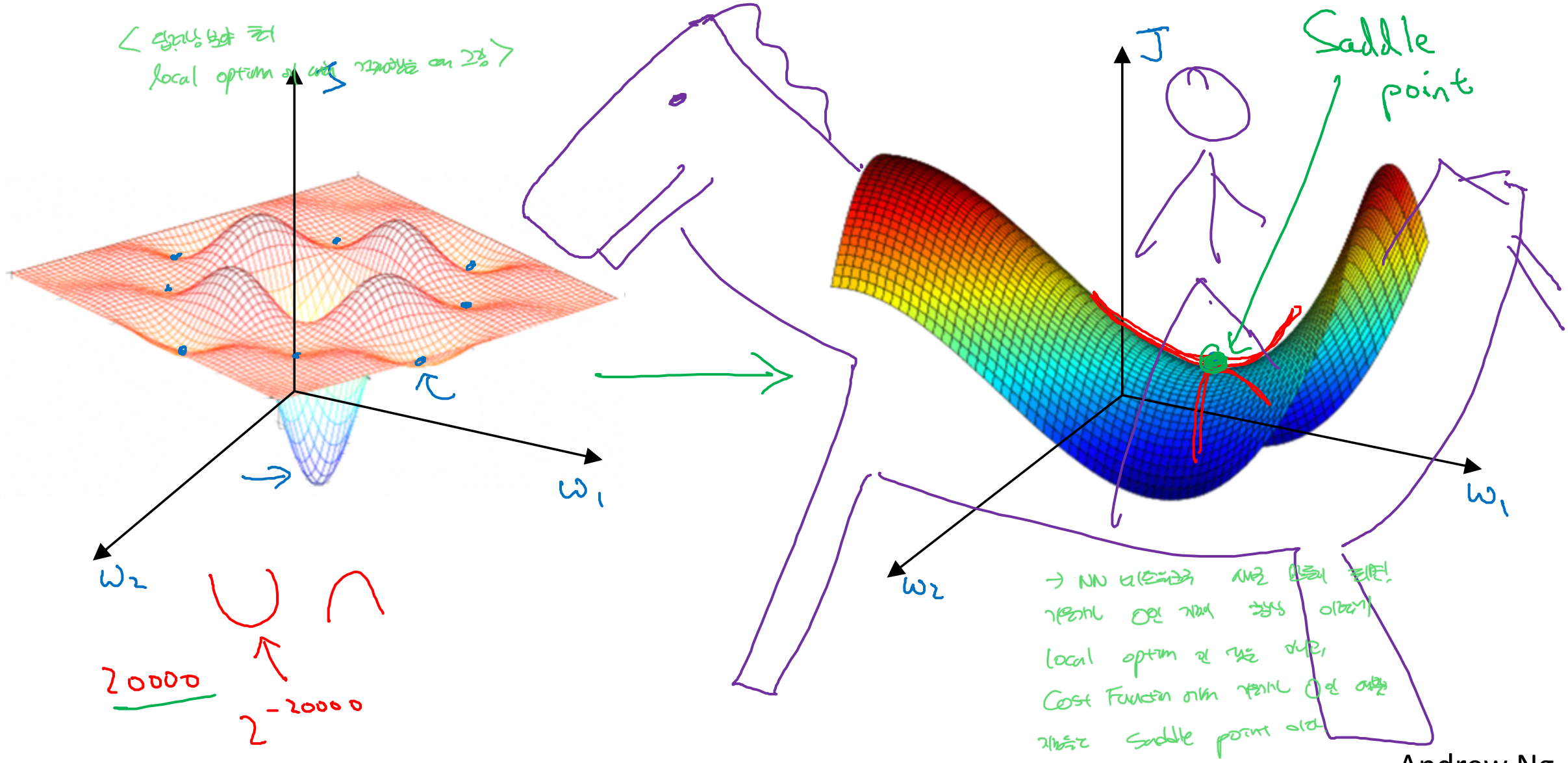
Manual decay (수작업으로 learning rate 조절 – 학습할 모델 수가 적을때 사용 가능)

deeplearning.ai

Optimization Algorithms

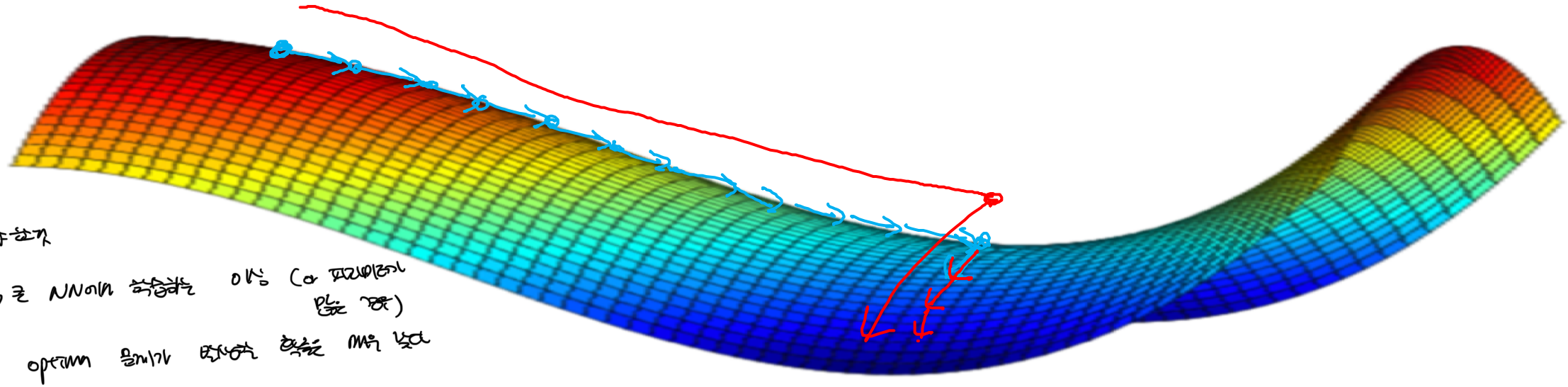
The problem of local optima

Local optima in neural networks



Andrew Ng

Problem of plateaus



- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow

Andrew Ng