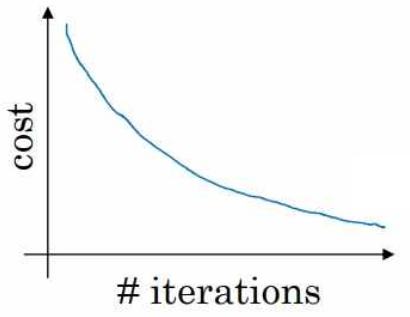
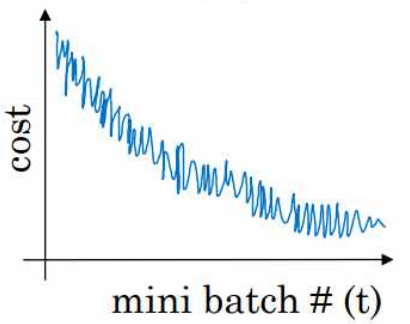
Optimization Algorithms

머신러닝을 적용한다는 것은 매우 경험에 의한 과정과 매우 반복적인 업무를 동반한다. 여러 모델들을 그냥 무조건 트레이닝시키고 가장 잘 작동하는 것을 찾아야 한다. 그렇기 때문에 빠르게 학습을 시킬 필요가 있다.
문제는 딥러닝이 큰 데이터에 가장 잘 작동하다는 것이다. 근데 큰 딥러닝 모델를 큰 데이테셋에 학습시키면 학습 속도는 매우 느릴 것이다. 이 때문에 학습을 빠르게 하고 좋은 최적화 알고리즘를 찾을 필요가 있다.

batch gradient descent에서 빠르게 cost function값 J를 최소화시킬 방법

1. Try better random initialization for the weights
2. try using Adam Optimizer
3. Try tuning the learning rate α
4. Try mini-bath gradient descent

Mini-batch gradient descent

Batch gradient descent $X,\ Y$	Mini-batch gradient descnt $X^{\{t\}},\ Y^{\{t\}}$
	

Vectorization allows to effieciently compute on m examples.

ex) $m = 5{,}000{,}000$

$$X_{(n_x,m)} = \left[\underbrace{x^{(1)}, x^{(2)}, \cdots, x^{(1000)}}_{\Rightarrow X^{\{1\}}_{(n_x,1000)}} \mid \underbrace{x^{(1001)}, \cdots, x^{(2000)}}_{\Rightarrow X^{\{2\}}_{(n_x,1000)}} \mid \cdots \mid \underbrace{\cdots X^{(m)}}_{\Rightarrow X^{\{5000\}}_{(n_x,1000)}}\right]$$

$$Y_{(1,m)} = \left[\underbrace{y^{(1)}, y^{(2)}, \cdots, y^{(1000)}}_{\Rightarrow Y^{\{1\}}_{(1,1000)}} \mid \underbrace{y^{(1001)}, \cdots, y^{(2000)}}_{\Rightarrow Y^{\{2\}}_{(1,1000)}} \mid \cdots \mid \underbrace{\cdots Y^{(m)}}_{\Rightarrow Y^{\{5000\}}_{(1,1000)}}\right]$$

mini-batch size = 1,000

number of mini_batches = 5,000

⇒ 5,000 mini-batches of 1,000 each

mini-batch t: $X^{\{t\}}$, $Y^{\{t\}}$ → $X^{(i)}$
$Z^{[l]}$
$X^{\{t\}}$, $Y^{\{t\}}$

in layer l when input is the example i from minibatch t.

$\Rightarrow Z^{[l]\{t\}(i)}$, $A^{[l]\{t\}(i)}$

repeat {

for $t = 1, \ldots, number\ of\ mini-batches$ {

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

$$Z^{[1]\{t\}} = W^{[1]} X^{\{t\}} + b^{[1]}$$
$$A^{[1]\{t\}} = g^{[1]}(Z^{[1]\{t\}})$$
$$\vdots$$
$$Z^{[l]\{t\}} = W^{[l]} A^{[l]\{t\}} + b^{[l]}$$
$$A^{[l]\{t\}} = g^{[l]}(Z^{[l]\{t\}})$$
$$\vdots$$
$$Z^{[L]\{t\}} = W^{[L]} A^{[L]\{t\}} + b^{[L]}$$
$$A^{[L]\{t\}} = g^{[L]}(Z^{[L]\{t\}})$$

Compute cost

$$J^{\{t\}} = \frac{1}{mini-batch\ size} \sum_{i=1}^{l} L(\hat{y}^{\{t\}(i)}, y^{\{t\}(i)}) + \frac{\lambda}{2*mini-batch\ size} \sum_{l} \| w^{\{t\}} \|_F^2$$

#Backprop to compute gradients cost $J^{\{t\}}$ using $X^{\{t\}}$, $Y^{\{t\}}$

$$W^{[l]} := W^{[l]} - \alpha\, d W^{[l]}$$
$$b^{[l]} := b^{[l]} - \alpha\, d b^{[l]}$$

}

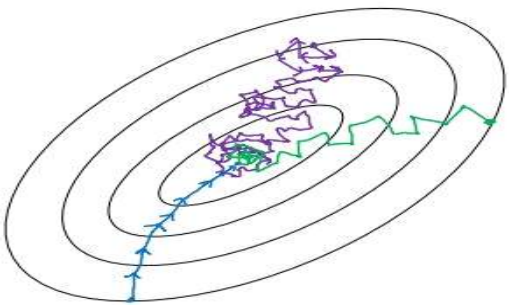
}

1 epoch: pass through training set

Choosing mini-batch size

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

If $mini-batch\,size = 1$: Stochastic gradient descent $(X^{\{t\}}, Y^{\{t\}}) = (x^{(i)}, y^{(i)})$

<table>
<tr><td colspan="3">in practice, in-between 1 and m</td></tr>
<tr><td>$mini-batch\,size = m$</td><td>in-between</td><td>$mini-batch\,size = 1$</td></tr>
<tr><td>Stochastic
gradient descent</td><td>mini-batch size
not to big/small</td><td>Batch gradient descent</td></tr>
<tr><td>lose speed up from vectorization</td><td>fastest learning
- Vectorization
- Make progress without processing entire training set</td><td>Too long per iteration</td></tr>
<tr><td rowspan="2">sgd는 절대 수렴하지 않으므로 최소값 근처 범위에서 있을 것이다. 절대 최소값에 도달하지 않는다.</td><td>최소값에 도달할 것이라고 보장할 순 없지만 sgd보다 조금 더 균등하게 최소값을 향한다.</td><td rowspan="2">process the whole training set before making progress.</td></tr>
<tr><td>항상 정확히 grad descent하거나 작은 범위에서 grad descent하지는 않는다.
이 문제라면 learning rate를 낮추면 된다.</td></tr>
<tr><td colspan="3">  </td></tr>
</table>

if small training set($m \leq 2{,}000$): Use batch gradient descent.

Typical mini-batch size:

컴퓨터 메모리에 따라 2의 지수값을 가질 때 더 빨리 실행된다.

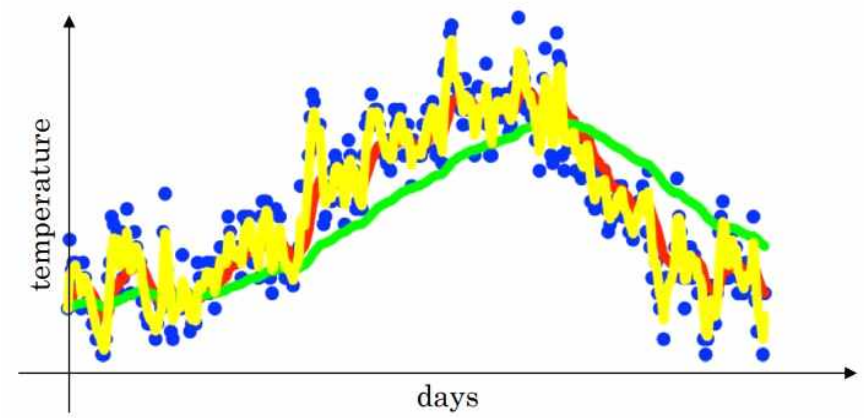
$2^6 = 64,\ 2^7 = 128,\ 2^8 = 256,\ 2^9 = 512,\ 2^{10} = 1024,\ \ldots$

⇒ make sure mini-batch fit in CPU/GPU memory

Exponentially weighted averages

$$v_t = \beta\, v_{t-1} + (1-\beta)\theta_t,\ \ v_0 = initial\ value$$

v_t: exponentially weighted average of $\frac{1}{1-\beta}$ points



- Increasing β will shift the line slightly to the right.
- Decreasing β will create more oscillation with in the line.

β일 때 첫 $\frac{1}{1-\beta}$동안은 $(1-\epsilon)^{\frac{1}{\epsilon}} \approx \frac{1}{e}$보다 큰 값이 이었다가 $\frac{1}{1-\beta}$ 이후에는 꽤 크게 감소할 것이다. $\frac{1}{1-\beta}$ 동안의 대략적인 평균이라고 볼 수 있다.

$$1-\beta = \epsilon \rightarrow \frac{1}{1-\beta} = \frac{1}{\epsilon}$$ (간단한 표현이고 정식적인 수학적 표현 아님)

$v_\theta = 0$

$v_\theta := \beta v_\theta + (1-\beta)\theta_1$

$v_\theta := \beta v_\theta + (1-\beta)\theta_2$

$\vdots$

$v_\theta = 0$

repeat {

get next θ_t

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

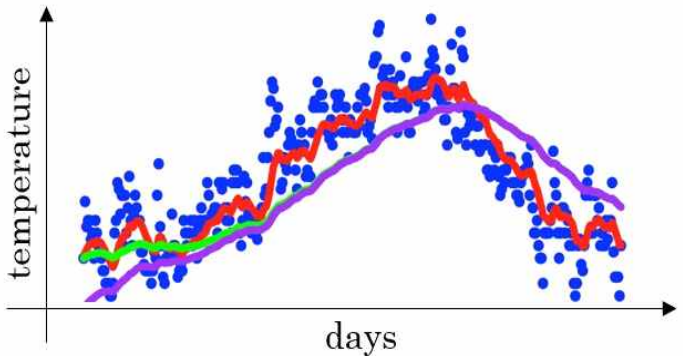
}

지수 가중 평균의 장점: 적은 메모리 양을 차지와 적은 코드의 효율성

산술 평균이 대개 더 좋은 추정치를 얻는다. 하지만 해당 기간의 모든 수치를 저장

하기 위해 더 많은 메모리가 필요하고, 구현도 더 복잡하고, 연산 부담도 더 크다. 딥러닝에서는 많은 양의 변수들의 평균값 계산이 필요하다. 지수 가중 평균은 연산과 메모리 소모에 매우 효율적이기 때문에 머신러닝에서 많이 사용된다.

Bias Correction in exponentially weighted average



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

if $\beta = 0.98$, $v_0 = 0$:

$v_1 = 0.98 v_0 + 0.02\theta_1$
$= 0.02\theta_1$

$v_2 = 0.98 v_1 + 0.02\theta_2$
$= 0.98 \times 0.02\theta_1 + 0.02\theta_2$
$= 0.0196\theta_1 + 0.02\theta_2$

초기 단계의 오류를 수정하는 방법

: 특히 추정의 초기 단계를 더 정확히 보정할 수 있다.

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

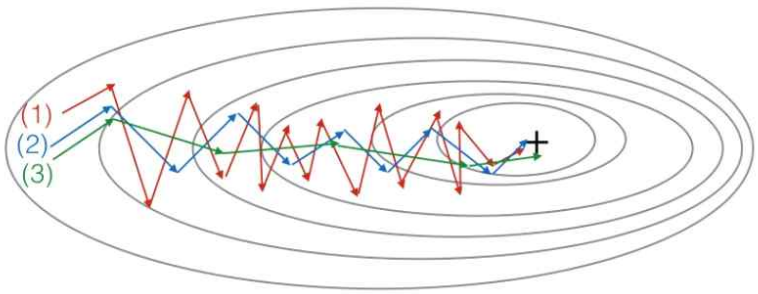
θ_t의 지수 가중 평균에 편향을 없앤 값

t가 충분히 커지면 β^t가 0에 가까워져서 편향 보정의 효과가 거의 없어진다.
t가 커질 때 보라색 곡선은 초록색 곡선에 가까워지는 이유이다.
bias correction은 보라색 선에서 초록색 선으로 가도록 한다.

지수 가중 평균을 구현하는 대부분의 경우 많은 사람들은 편향 보정을 거의 구현하지 않는다. 초기 단계를 그냥 기다리고 편향된 추정이 지나간 후부터가 관심 범위이기 때문이다. 하지만 초기 단계의 편향이 신경 쓰인다면 편향 보정은 초기에 더 나은 추정값을 얻는 데 도움이 될 것이다.

Gradient descent with Momentum

gradient descent보다 일반적으로 더 빨리 작동한다. 기본 아이디어는 기울기의 기하급수적 가중 평균치를 산출하는 것이다. 그리고 이 기울기를 이용해 가중치를 업데이트하는 것이다.



상하로는
learning slower

좌우로는
learning faster

Momentum: $V_{dW}=0,\ V_{db}=0$

on iteration t:

Compute dW, db on current min-batch

$V_{dW}=\beta_1 V_{dW}+(1-\beta_1)dW$

$V_{db}=\beta_1 V_{db}+(1-\beta_1)db$

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

similar: β_1 = friction/ V_{dW}, V_{db} = velocity/ dW, db = accelerator

β_1가 클수록 부드럽게 gradient descent 한다.

(1): gradient descent

(2): gradient descent with momentum(small β_1)

(3): gradient descent with momentum(large β_1)

지수 가중 평균을 이용하여 gradient descent를 조금 더 부드럽게 할 수 있다. 세로 방향을 더 느리게 하여 세로 방향의 변동, 즉 세로 방향의 평균은 거의 0에 가깝게 한다. 가로 방향에서 모든 derivate가 (위 그림의 좌측에서 시작한다고 했을 때) 가로 방향의 오른쪽으로 가로 방향의 변동, 즉 가로 방향의 평균은 꽤 큰 값이다.
이것이 알고리즘이 더 단순하게 gradient descent를 해주거나 최소값의 방향에서 변동이 무뎌지게 해준다.

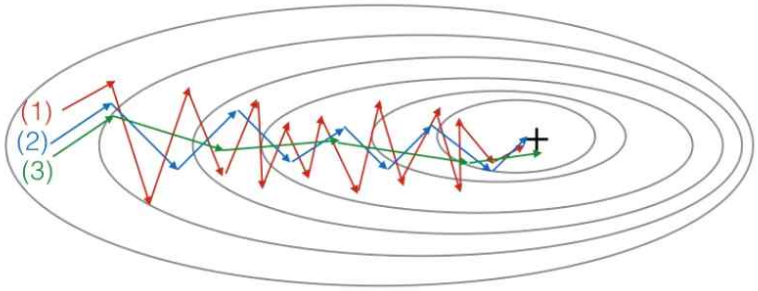
gradient descent와 momentum에 대한 학술을 보면 $(1-\beta_1)$가 생략되어 있는 $V_{dW}=\beta_1 V_{dW}$와 $V_{db}=\beta_1 V_{db}$를 볼 수 있다.

효과는 V_{dW}, V_{db}가 $(1-\beta_1)$로 스케일이 된다는 것이다. 더 정확히 말하면 $\frac{1}{1-\beta_1}$로 말이다. grad를 update할 때 learning rate α와 그에 상응하는 $\frac{1}{1-\beta_1}$로 변해야 한다.

실제로는 $(1-\beta_1)$가 있든 없든 잘 작동한다. learning rate α가 가장 최적의 값이 어떤 것인지가 영향을 준다. 그러나 β_1를 튜닝하면 V_{dW}와 V_{db}에 모두 영향을 주기 때문에 스케일에도 영향을 준다. 결과적으로 learning rate α를 다시 튜닝해야 할 수 있다. 이 때문에 $(1-\beta_1)$를 포함하는 공식을 선호한다.

$\beta_1 = 0.9$일 때 두 표현 모두 흔한 방법이다. 단지 learning rate α를 두 방법에서 다르게 설정해야 하는 점이 다르다.

RMSprop(Root Mean Square propagation)



상하로는
learning slower

좌우로는
learning faster

$S_{dW}=0,\ S_{db}=0$

on iteration t:

Compute dW, db on current min-batch

$$S_{dW}=\beta_2 S_{dW}+(1-\beta_2)(dW)^2$$

$$S_{db}=\beta_2 S_{db}+(1-\beta_2)(db)^2$$

$$W:=W-\alpha\frac{dW}{\sqrt{S_{dW}}+\epsilon},\ b:=b-\alpha\frac{db}{\sqrt{S_{db}}+\epsilon}$$

β_2가 클수록 부드럽게 gradient descent 한다.

(1): gradient descent

(2): gradient descent with RMSprop(small β_1)

(3): gradient descent with RMSprop(large β_1)

가로 방향의 업데이트를 빠르게 하고 싶기 때문에 $(dW)^2$의 값은 작아야 하고 세로 방향의 업데이트는 느리게 하고 싶기 때문에 $(db)^2$의 값은 커야 한다.

큰 α 값을 사용하면 세로 방향을 덜 갈라지면서 더 빠른 러닝을 진행한다.

S_{dW}의 값이 작을수록 W는 빠르게 업데이트 된다.

ϵ은 분모가 0이 되는 것을 방지한다. S_{dW}와 S_{db}가 0에 가까워 $\frac{dW}{\sqrt{S_{dW}}}$와 $\frac{db}{\sqrt{S_{db}}}$가 폭발적으로 커지는 것을 방지하기 위해 아주 작은 ϵ 값을 더해준다.

ϵ은 어떤 값이든 상관 없다. 10^{-8}이 합리적인 기본값이다. 10^{-8}는 반올림이나 다른 이유에서 조금 더 안정적인 수치를 제공한다.

RMSprop은 momentum과 비슷하게 gradient descent나 mini-batch gradient descent에서 변동을 무디게 하는 효과가 있다. 조금 더 큰 learning rate α를 사용하여 알고리즘의 러닝 속도를 높힐 수 있다.

Adam optimization algorithm

여러 최적화 알고리즘이 훈련을 하고자 하는 넓은 범위의 신경망에서 일반적으로 잘 작동하지 않았다. 그래서 딥러닝 커뮤니티에서는 새로운 최적화 알고리즘에 약간의 의심을 갖게 되었다. 모멘텀이 있는 경사하강법이 아주 잘 작동하기 때문에 더 잘 작동하는 알고리즘을 제안하기 어려운 것도 있었다. RMSprop과 Adam optimization은 넓은 범위의 딥러닝 아키텍처에서 잘 작동하는 알고리즘으로 자리 잡았다. 망설이지 않고 시도해도 좋은 알고리즘이다.

Adam(Adaptive moment estimation) Optimization algorithm은
RMSprop과 Momentum을 합친 알고리즘이다.

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

on iteration t:

Compute $dW,\ db$ on current min-batch

$V_{dW}=\beta_1 V_{dW}+(1-\beta_1)dW,\ V_{db}=\beta_1 V_{db}+(1-\beta_1)db$

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

$$V_{dW}^{corrected}=\frac{V_{dW}}{1-\beta_1^t},\ V_{db}^{corrected}=\frac{V_{db}}{1-\beta_1^t}$$

$$S_{dW}^{corrected}=\frac{S_{dW}}{1-\beta_2^t},\ S_{db}^{corrected}=\frac{S_{db}}{1-\beta_2^t}$$

$$W:=W-\alpha\frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected}}+\epsilon},\ b:=b-\alpha\frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}}+\epsilon}$$

전형적인 Adam optimizer 구현에서는 bias correction을 한다.

Hyperparameters choice:

α: need to be tune

β_1: 0.9 → $dW,\ db$

β_2: 0.999 → $(dW)^2,\ (db)^2$

ϵ: 10^{-8}

Adam 논문의 저자가 추천하는 값으로

이 중에 ϵ은 크게 상관 없다. 값을 설정하지 않아도 전체 성능에는 영향이 없기 때문이다.

일반적으로 Adam을 구현할 때 β_1과 β_2, ϵ을 모두 기본값을 사용한다.

이 셋은 보통 건드리지 않고 α에 여러 값을 시도해 가장 좋은 값을 찾는다.

Learning rate decay

slowly reduce α

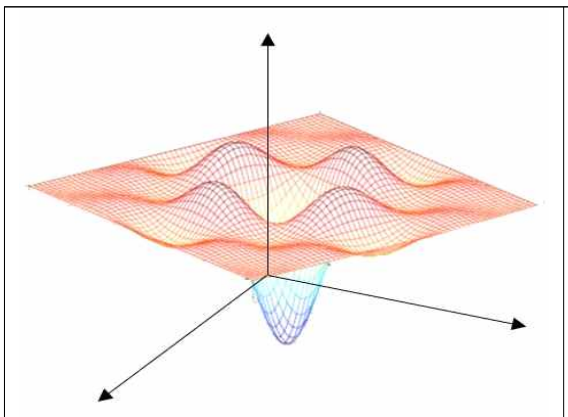
α가 작이지면 단계마다 진행 정도가 작아지고 최솟값 주변의 밀집된 영역에서 진동할 것이다. 훈련이 계속되면서 최솟값 주변에 배회하는 대신.

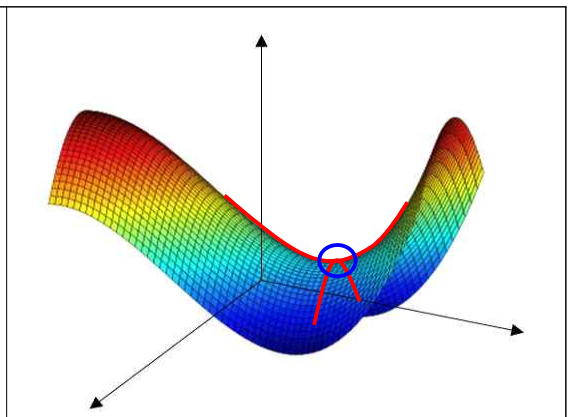
α를 서서히 줄인다는 것은 학습 초기 단계에서는 훨씬 큰 스텝으로 진행하고, 학습이 진행할수록 정도가 느려져 작은 스텝으로 진행한다.

1. $\alpha = \frac{1}{1 + decaying\_rate \times epoch\_num}\alpha_0$
2. $\alpha = 0.95^{epoch\_num}\alpha_0$: exponentally decay
3. $\alpha = \frac{k}{\sqrt{epoch\_num}}\alpha_0$ or $\alpha = \frac{k}{\sqrt{t}}\alpha_0$
4. discrete staircase: 이산적 단계로 감소하는 learning rate
5. manual decay: 작은 수의 훈련 데이터를 가진 모델의 경우에만 가능

The problem of local optima

초기 딥러닝 학자들은 최적화 알고리즘 local optima에 걸리는 것에 대해 걱정했다. 하지만 딥러닝 분야가 발전해가면서 local optima에 대한 이해가 날로 변하고 있다. 현재 시점에서 딥러닝이 local optima는 큰 문제가 아니다.





비공식적으로 고차원 공간에서 정의되는 함수에서 기울기가 0인 경우 방향에 따라서 convex ligh 함수이거나 concave light 함수일 수 있다.

예를 들어 20,000차원의 공간의 경우 local optima가 존재하기 위해서는 모든 20,000가지의 방향이 아래로 볼록한 모양 ∪처럼 생겨야 한다. 그렇게 될 확률은 매우 작은데 아마 2^{-20000}일 것이다.

반대로 위로 볼록한 모양 ∩ 혹은 오른쪽 그래프에서 두 빨간색 선 같은 경우가

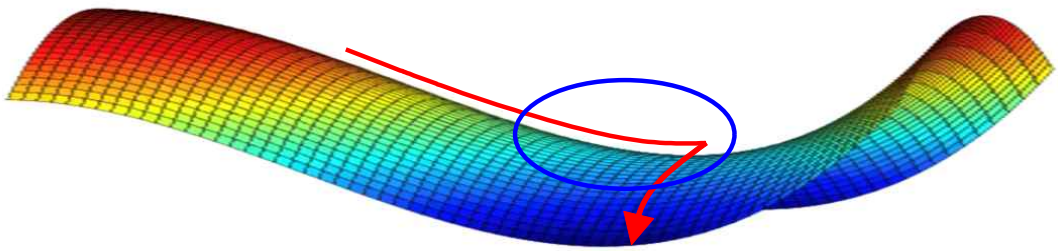
더 많을 것이다. 그렇기 때문에 local optima 보다 고차원의 공간에서는 그림과 같이 saddle point에 접할 확률이 매우 높다.

신경망 네트워크를 만들면 기울기가 0인 점이 항상 local optima인 것은 아니다. 대신 비용함수에서 기울기가 0인 대부분 점들은 saddle point이다.

Problem of Plateaus

local optima가 문제가 되지 않는다고 결론 지을 수 있었다. 그런데 saddle poiunt의 plateau가 러닝 속도를 저하시킬 수 있다.

plateau는 함수 기울기 값이 0에 근접한 긴 범위이다.



- Unlikely to get stuck in a bad local optima

비교적 큰 신경망 네트워크를 학습시키거나 파라미터가 많은 경우 local optima에 갇힐 확률은 낮다. 비용함수 J는 비교적 고차원 공간에서 정의된다.

- Plateaus can make learning slow

Plateau가 러닝 속도를 늦추게하는 요소로 문제이긴 하다. Momentum 또는 RMSprop의 Adam과 같은 알고리즘이 plateau를 빠져나오는 속도를 높혀줘서 plateau 구간을 완전히 빠져나오는데 도움을 준다.