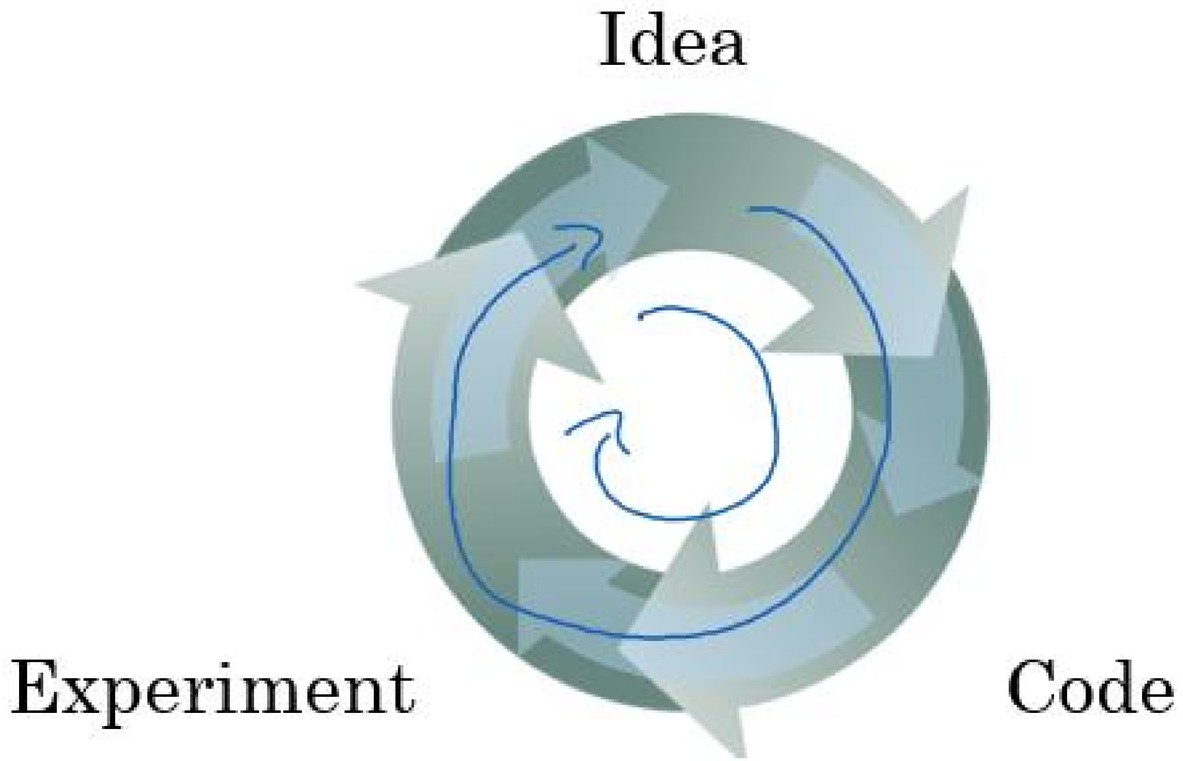
2nd Course:
Improving Deep Neural Networks

- Hyperparameter tuning
- Regularization
- Optimization
- Stochastic Gradient Descent(Mini-bach)
- Momentum

Week1

- 어떻게 Neural Network(NN)을 잘 작동하게 할 수 있는 방법에 관한 내용 학습
- 해당 발표에서는 수식의 설명보다는 왜 사용하는지 이유에 관해서 발표 진행!

Hyperparameter Tuning

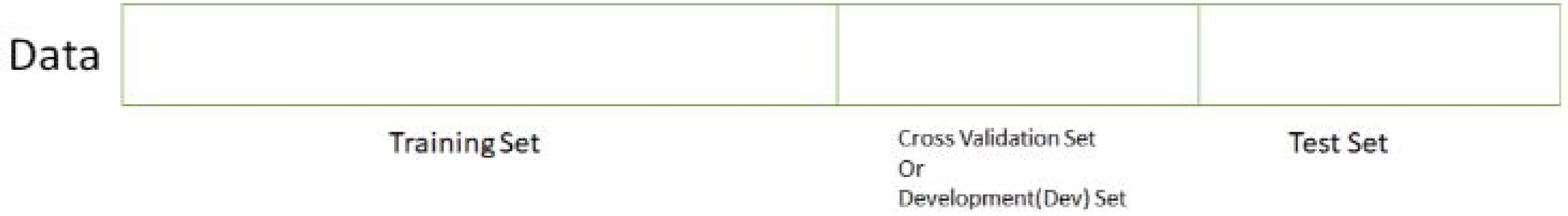


Applied ML is a highly iteration process

#layer, #hiddne unit, Learning Rate(alpha), Activation Function ...
적절한 하이퍼 파라미터 값들을 바로 선택하기는 불가능해서 반복적인 과정이 필요

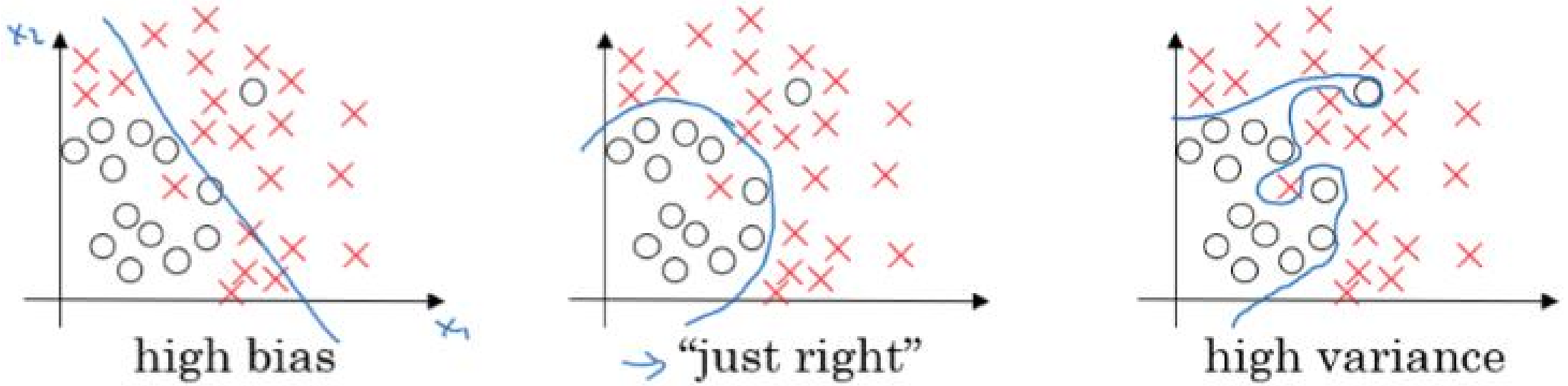
Data Setting

그러면 반복적인 과정을 얼마나 효율적으로 하느냐가 중요한 Key point!



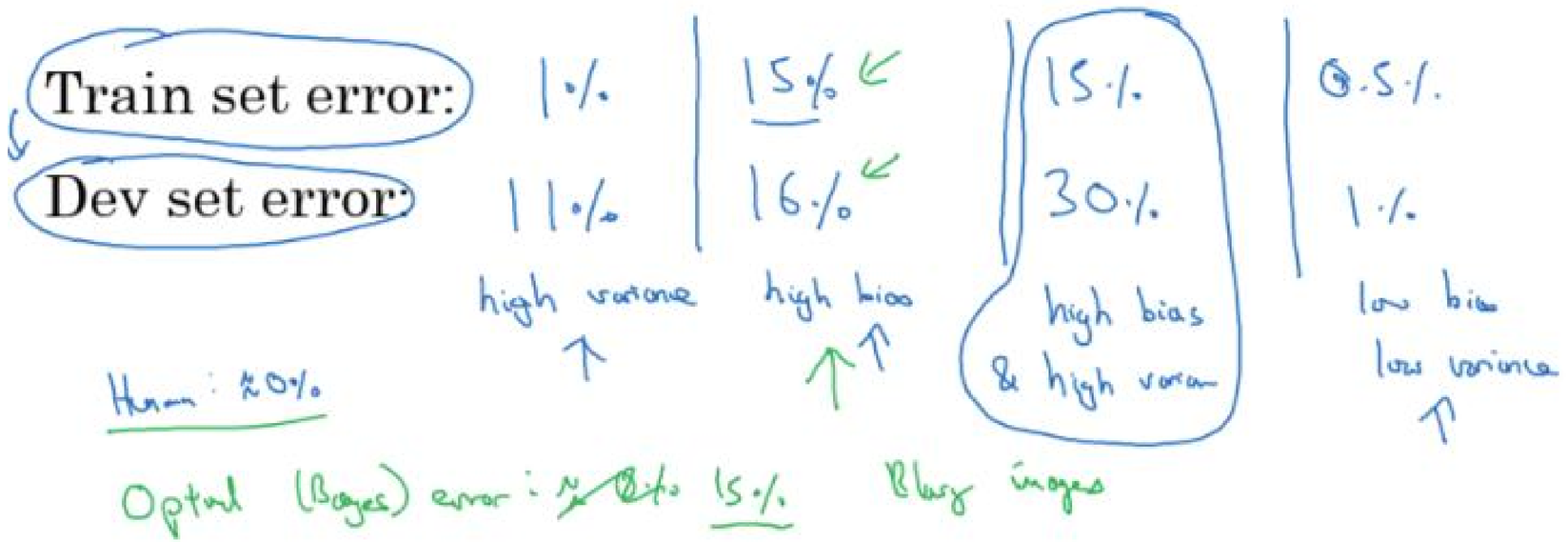
위와 같이 Train / Val(Dev) / Test를 나누면 Bias, Variance 문제를 더 효율적으로 계산하고 개선할 수 있다

Bias / Variance



- high bias : 데이터가 잘 fitting 되지않고 편향된 값 다수 존재
- just right : good!
- high variance : 데이터를 완벽하게 fitting 시키다보니 overfitting 존재

Bias / Variance



- first example : Train good, but Dev bad → high variance! Train에 너무 초점
- second example : Train Bad → high bias(편향된 값 다수 존재)

Bias / Variance

제일 큰 문제점인 hight bias / variance 일 경우는 어떻게 해결?

- high bias : 더 큰 network나 다른 구조의 neural network 선정, 학습 시간 늘리기, 최적화 알고리즘 사용
- high variance : 더 많은 데이터 수집, **정규화(regularization)**

Regularization

L2 regularization(Logistic Regression)

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

L2 regularization(Neural Network)

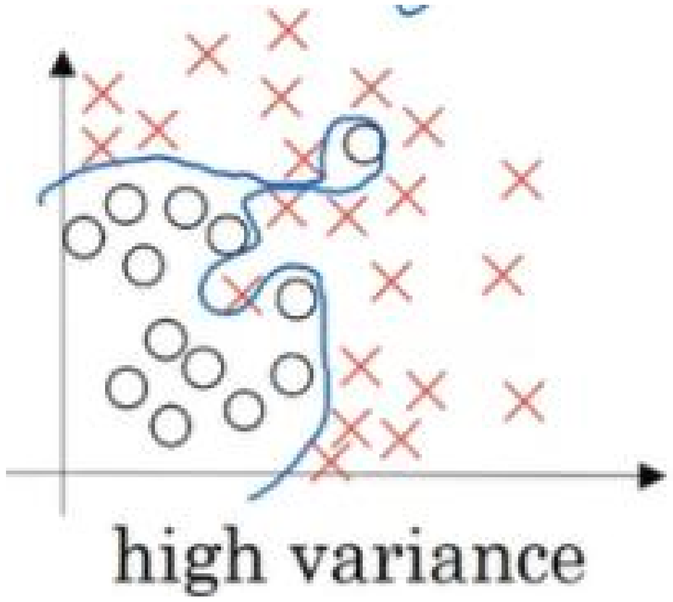
$$J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m}\mathscr{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|W^{[l]}\|^2$$

L2 regularization(Gradient)

$$\begin{aligned} W^{[l]} &:= W^{[l]} - \alpha\left[(\text{from BP}) + \frac{\lambda}{m}W^{[l]}\right] \\ &= W^{[l]} - \frac{\alpha\lambda}{m}W^{[l]} - \alpha(\text{from BP}) \\ &= (1 - \frac{\alpha\lambda}{m})W^{[l]} - \alpha(\text{from BP}) \end{aligned}$$

Regularization

그럼 왜 정규화가 오버피팅을 줄여주나?



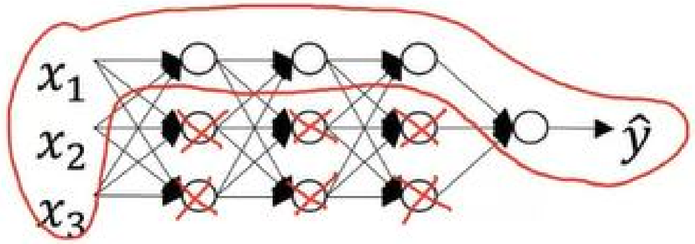
overfitting!

Cost function

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

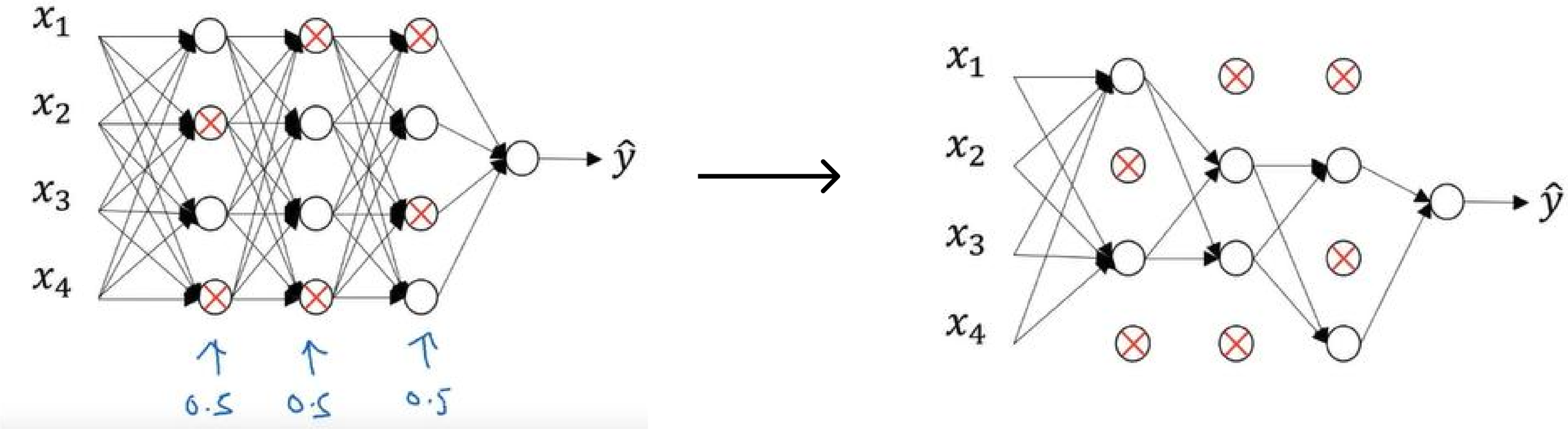
이때 L2가 람다를 크게 설정하면 매개변수인 W(가중치) 거의 0에 가깝게 만들 수 있어, 곧 hidden unit들의 영향력을 줄여 단순한 신경망으로 만든다!

↓



Dropout

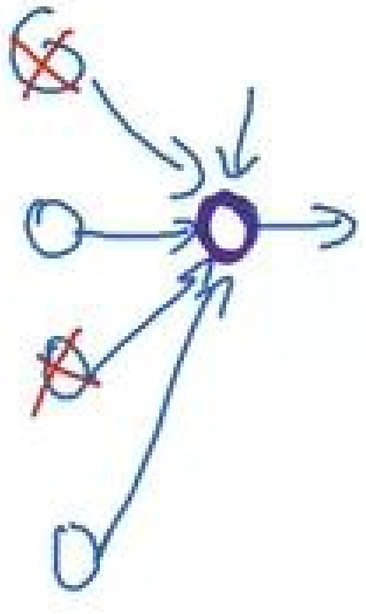
L2 Regularization은 가중치를 줄이는 거라면, Dropout은 노드 삭제!



- 설정한 keep_prob은 유지 확률, (1 - keep_prob)은 제거 확률
- 주의해야 할 점은 test set에서는 dropout 사용 x
- 예측값이 랜덤하게 되고, 결국에 예측값에 noise만 추가될 뿐!

Dropout

무작위 unit 제거가 어떻게 Regularization을 잘 작동?



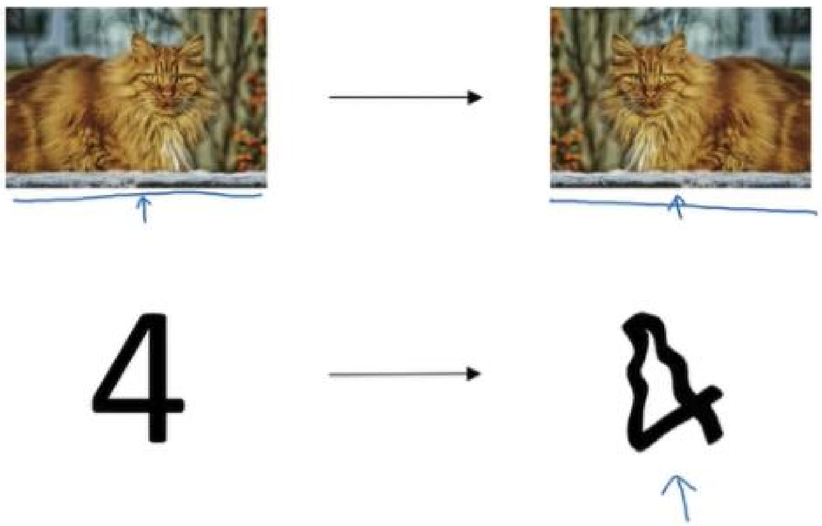
- dropout을 도입하게 되면 unit이 제거될 수 있는데 결국 그러면 보라색 노드는 특정 feature(왼쪽 노드 4개)에 의존해서는 안된다.
- 한 특성에 큰 가중치를 주지 않고, 분산시키게 되어 L2 Regularization과 유사하게 weight를 줄이고 overfitting 문제 해결!

주의사항

- 입력값이 크기가 큰 CV(Computer vision)에서 많이 사용된다.
- overfitting 방지에 도움을 주는 것이므로 overfitting이 아니면 사용 x

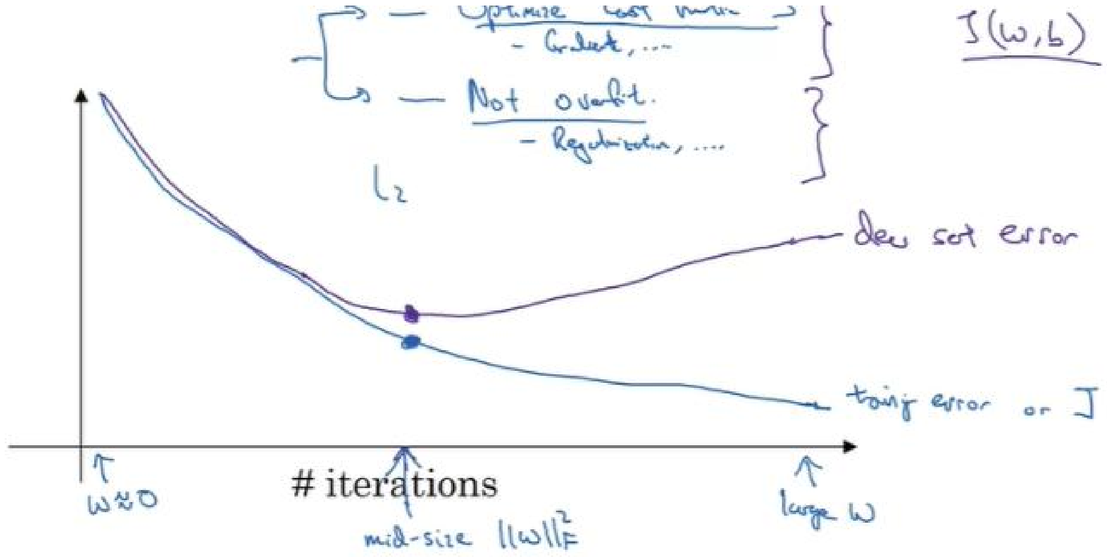
Data augmentation and Earlhy Stopping

Data augmentation



high variance를 해결하는 과정 중 더 많은 데이터 수집하는 방법(추가적인 비용 발생 x)

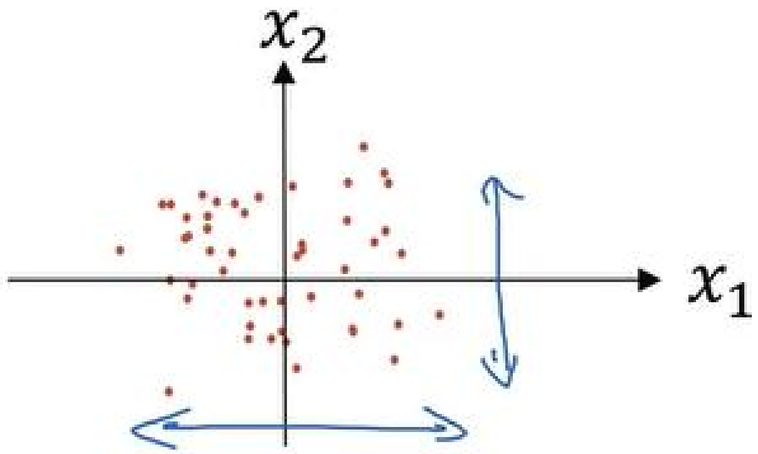
Early Stopping



- train set error, dev set error 그래프에서 W가 middle-size일 때 중지
- 머신러닝의 두 단계(cost function 최적화, not overfit)를 함께 적용하여 문제 각각 해결 할 수 없는 문제점 존재 → L2 Regularization 선호

Optimization - Normalizaing inputs

학습속도를 높이기 위한 입력데이터 표준화(Normalization)

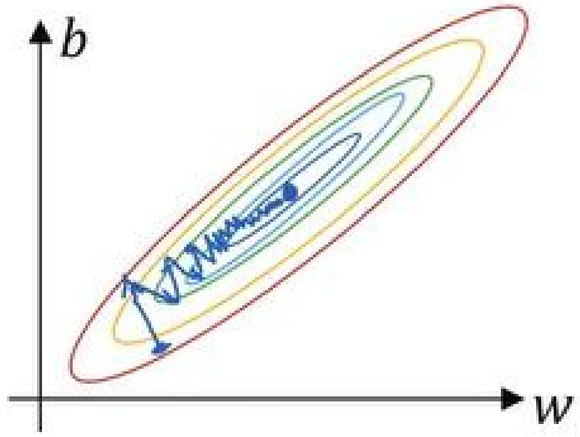


1. 평균을 빼거나, 0으로 만드는 방법
2. Normalize variance(분산 표준화)

위 방법을 사용하여 최종적으로 왼쪽과 같은 입력데이터 표준화

Optimization

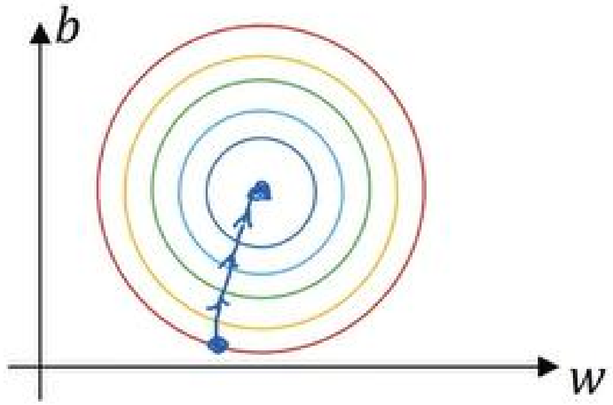
Unnormalized



- unnormalized feature 사용한다면 매우 작은 learning rate 사용
- Gradient Descent 수행하면 더 많은 단계 거쳐서 최솟값 도달 (진동이 많음)

vs

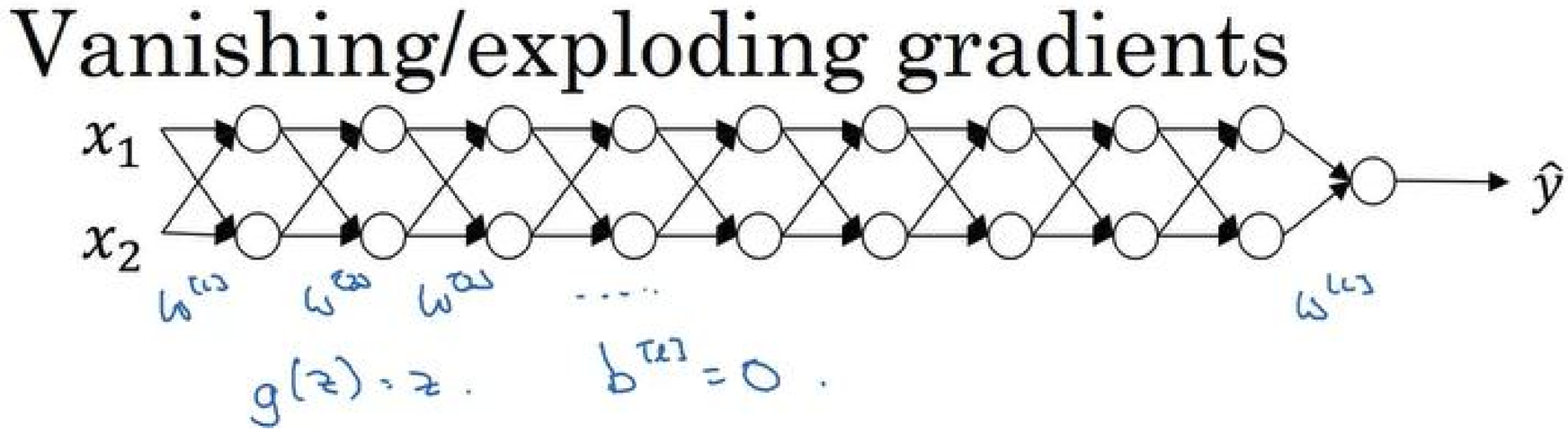
Normalized



- normalized feature 사용한다면 구형의 모양으로 존재
- Gradient Descent 수행하면 어디서 시작하더라도 바로 최솟값 도달 가능

Vanishing / Exploding gradients

DNN(Deep Neural Network) 학습할 때 가장 큰 문제점 중 하나는
기울기 매우 작아지거나 증가하는 경우!



$$\hat{y} = W^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} x$$

- 결과값은 다음과 같은데, 결국 결과값에 1.5^L을 곱해지므로 매우 deep한 NN일 경우 결과값 y^hat 기하급수적으로 증가
- 1.5를 0.5로 변경하면 0.5^L이 되어서 결과값은 매우 작은 값
- 즉 W의 비중과 L에 따라서에 따라서 activation unit이 매우 커지거나 작아진다.

Weight Initialization

vanishing / exploding gradients를 해결하기 위해서 초기화를 잘 해주자!(initialization)

- Random Initialization(feature 수인 n이 많을 수록, w는 더 작아지게 설정)

1. 일반적, tanh

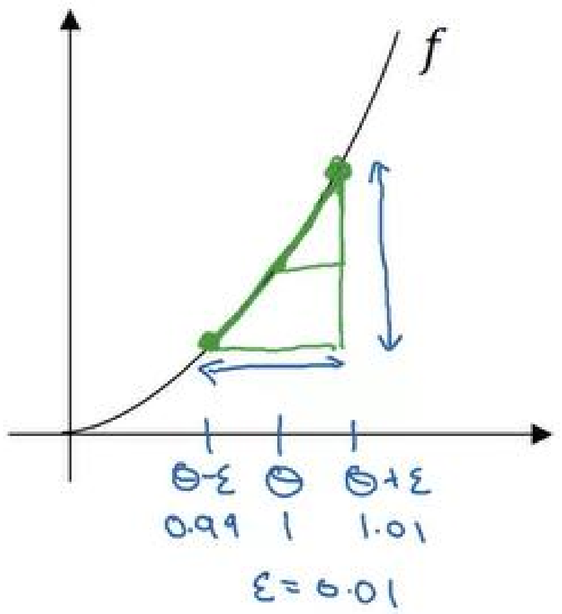
$W^{[l]}$ = np.random.randn(shape) * np.sqrt($\frac{1}{n^{[l-1]}}$)

2. ReLU

$W^{[l]}$ = np.random.randn(shape) * np.sqrt($\frac{2}{n^{[l-1]}}$)

Numerical approximation of gradients

Back propagation 사용하는 경우에 Gradient Checking 통해 Back propagation 테스트 진행 가능



- 번외 : 미분학에서의 기울기 공식

$$f'(\theta) = \lim_{\epsilon \to 0} \frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon}$$

큰 삼각형

- 기울기(높이 / 너비)

$$\frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon} \approx g(\theta)$$

→ 3.0001

- 미분

$$f'(\theta) = g(\theta) = 3\theta^2$$

→ 3

- 오차 : 0.0001

작은 삼각형

- 기울기(높이 / 너비)

$$\frac{f(\theta+\epsilon) - f(\theta)}{\epsilon}$$

→ 3.0301

- 미분

$$f'(\theta) = g(\theta) = 3\theta^2$$

→ 3

- 오차 : 0.0301

오차를 통해 확인해본 결과 미분값의 근사치는 two-sided difference(큰 삼각형)이 미분값과 더 유사해서 정확도가 높다

Gradient Checking

1. 정확한 dtheta를 계산했는지 알아보기 위해 gradient checking 도입

$$dtheta = \frac{\partial J}{\partial \theta} = x.$$

2. gradient approx, 즉 추정치를 구하기

1. $\theta^+ = \theta + \varepsilon$
2. $\theta^- = \theta - \varepsilon$
3. $J^+ = J(\theta^+)$
4. $J^- = J(\theta^-)$
5. $gradapprox = \frac{J^+ - J^-}{2\varepsilon}$

3. 비교값 확인

$$difference = \frac{\| grad - gradapprox \|_2}{\| grad \|_2 + \| gradapprox \|_2}$$

$\epsilon = 10^{-7}$로 설정

→ 비교값보다 작으면 good
10^-5 : 괜찮지만 다시 확인
10^-3 : 다시 점검

Gradient Checking Tip

1. Don’t use in training - only to debug
   - 계산 시간이 오래걸리기 때문에 디버깅 용도로만 사용
2. If algorithm fils grad check, look at components to try to identify bug
   - 알고리즘이 Grandient Check 실패하면 값이 크게 다른 i를 찾아서 버그 트래킹 가능
3. Remember regularization
   - Regularization 항이 있다면 dtheta에 추가
4. Doesn’t work with dropout
   - dropout에서는 unit을 임의로 제거하기 때문에 동작 x

Mini-batch Gradient Descent

알고리즘이 더 빠르게 학습할 수 있도록 도와주는 최적화 알고리즘

Gradient Descent

- 파라미터를 업데이트할 때 모든 Training example “m”에 대해서 gradient 계산한 후 파라미터 업데이트
- 이는 결국 m이 매우 커질수록 느려지가 된다. (벡터화를 통해서 효율적으로 할 수 있지만..)

vs

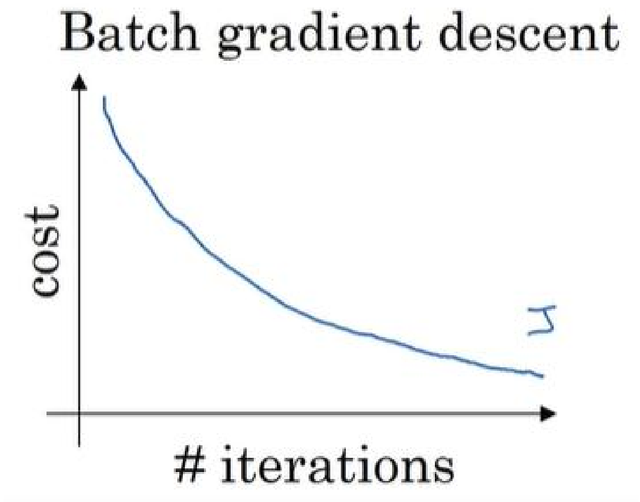
Mini-batch Gradient Descent

- 모든 training example m 에 대해서가 아니라 중간에 파라미터 업데이터하여 더 빠른 알고리즘 진행 가능
- m = 5,000,000 → 1,000개 나눔 → x^{1} 결국 mini-batch는 총 5,000개의 세트다.
- 정리 : mini-batch t : X^{t}, Y^{t}

mini-batch GD는 mini-batch를 한 번에 처리해서 파라미터를 업데이트 한다는 의미!

Mini-batch Gradient Descent

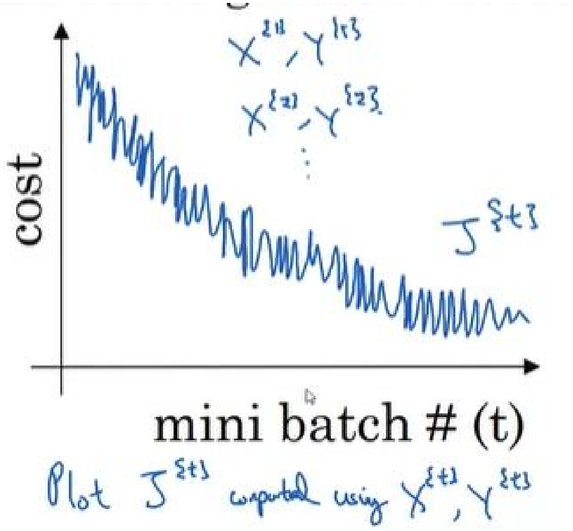
Gradient Descent



- 잔잔한 곡선 모양
- cost가 계속 감소

vs

Mini-batch Gradient Descent



- 매 반복마다 X^t, Y^t를 처리
- 결국 반복마다 다른 training set 학습진행하는 것과 동일
- Mini-batch GD는 noisy한 모양

Mini-batch Gradient Descent

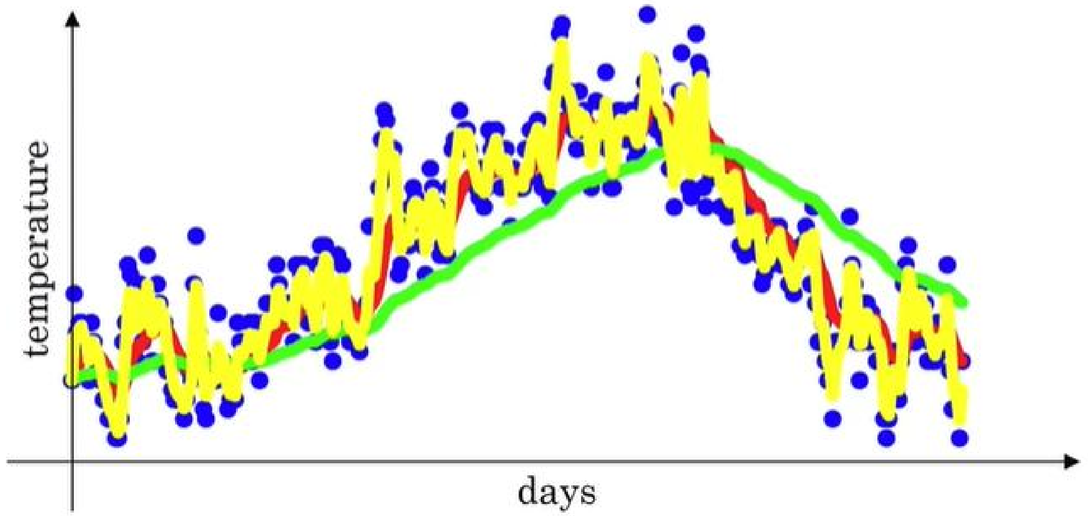
그러면 1 ~ m의 값 중 mini-batch size는 어떤 것을 선택해야할까?

1. Training set의 크기가 작다면, batch GD 사용 → m의 크기가 작다면 mini-batch GD 의미 x
ex) n <= 2000

2. m이 충분히 크다면, mini-batch size는 64 ~ 512가 보편적
2의 지수를 사용하는 이유는 컴퓨터 메모리의 형식과 비슷하여, 더 빨리 실행된다.
결국 batch_size도 hyper-parameter!

3. 모든 mini-batch X^t, Y^t가 CPU / GPU memory에 있도록 한다.
CPU / GPU에 들어가지 않는 mini-batch를 처리하는 경우에 성능이 저하되고, 더 악화된다.

Exponentially Weighted Averages



$v_t = \beta v_{t-1} + (1-\beta)\theta_t$ $\frac{1}{1-\beta}$ days의 평균 기온

- Red : B = 0.9 → average : 10
- Green : B = 0.98 → average : 50
- Yellow : B = 0.5 → average : 2

Beta가 작으면 noisy한 형태이고, 변화는 빨리 반영하지만 이상치에 민감

정리

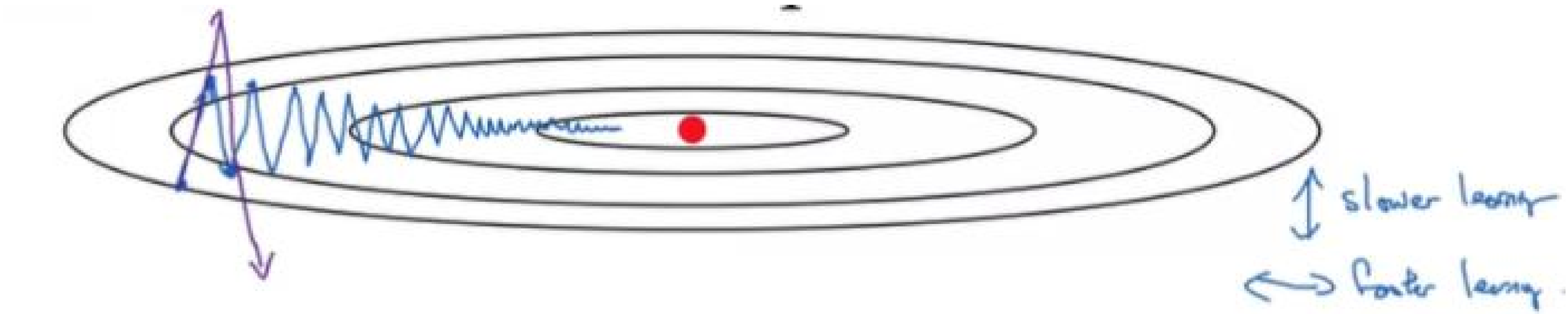
gradient의 변화량을 측정할때, 지속적으로 가산해 임의로 학습을 중단시키는 것이 아니라, 지역적으로 gradient를 취합해서, 학습정도를 조절해주는 역할을 한다.

장점

구현 과정에서 하나의 v_theta를 가지고 갱신하면서 EMA를 구할 수 있기 때문에 기본적으로 단 한 줄의 코드로 단 하나의 값만 메모리에 저장하면 되기 때문에 아주 적은 양의 메모리가 사용되고, 효율성이 높다

Gradient Descent with momentum

momentum는 GD의 EMA를 구하고, 이 값을 이용해서 파라미터 weight 업데이트하는 것



1. Cost function을 최적화시키려고하면 파란색 그래프처럼 최소값을 향해 진동을 하면서 접근
2. 진동폭이 GD를 느리게하는데, 훨씬 더 큰 Learning rate를 사용하는 경우 보라색 그래프처럼 발산 가능성 존재
3. 진동폭을 줄이기 위해 세로축은 slow learning, 가로축은 fast learning 도입
4. 이것은 momentum을 통해 감소시킬 수 있다!

Gradient Descent with momentum

On iteration t:

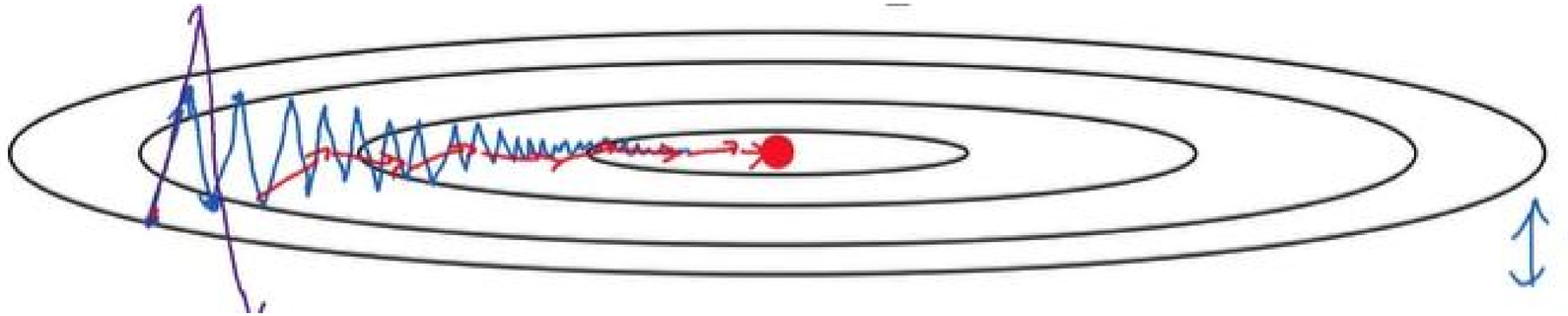
Compute dW, db **(on current mini-batch)**

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

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

$$W = W - \alpha V_{dW}$$

$$b = b - \alpha V_{db}$$



1. mini-batch에서 dW, db를 구한다.
2. dW, db에 대해서 EMA를 구한다.
3. 구한 EMA를 가지고 W, b 업데이트
4. 이 과정을 통해 GD는 smooth하게 된다.

결과적으로 세로축의 변동 평균은 거의 0이 되고
가로축의 변동 평균이 꽤 큰 값이 되어서
빨간색 그래프처럼 최소값을 향해서 접근!