Deep Learning

Đỗ Trọng Hợp
Khoa Khoa Học và Kỹ Thuật Thông Tin
Đại học Công Nghệ Thông Tin TP. Hồ Chí Minh

Improving Deep Neural Network

Trong-Hop Do

Setting up your ML application

Parameters vs Hyperparameters

- Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \ldots$

- Hyperparameter:

  Learning rate α

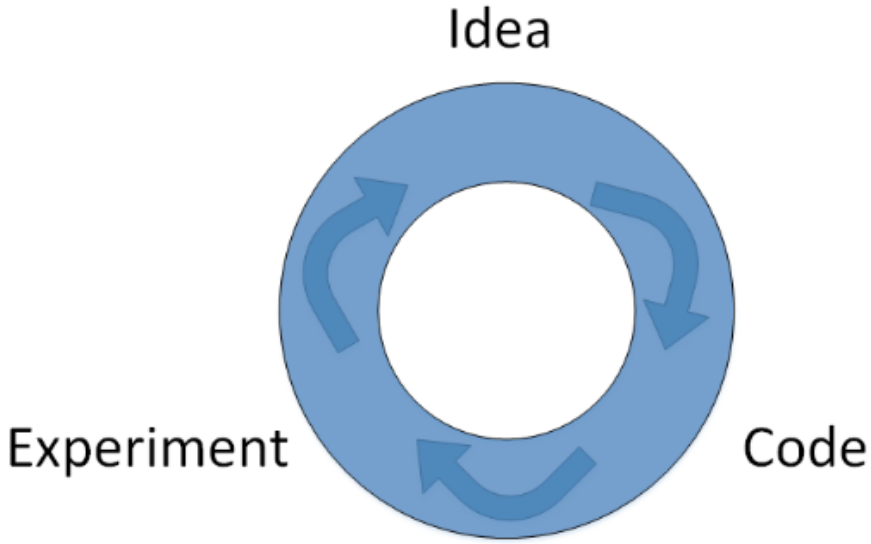
  # iterations

  # hidden layers L

  # hidden units $n^{[1]}, n^{[2]}, \ldots$

  Choice of activation function

  Momentum, minibatch size, regularizations,…



Applied ML is a highly iterative process

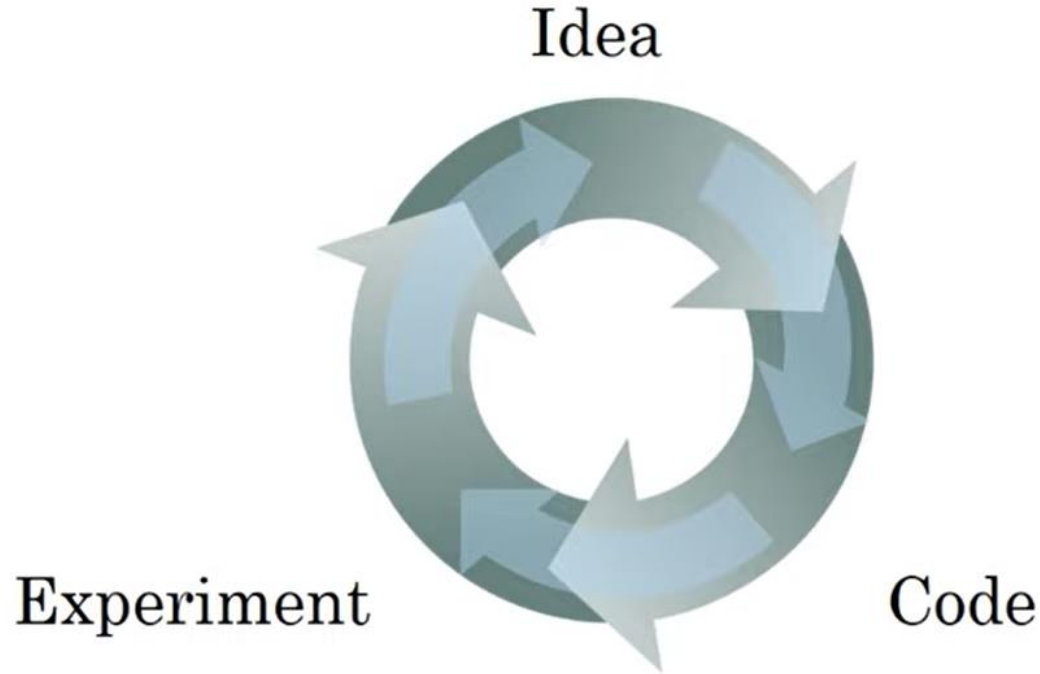
layers

hidden units

learning rates

activation functions

…



Train/Dev/Test Sets

Development set: used to check the network performance to make necessary changes

Test set is used to get unbiased estimation of the network performance

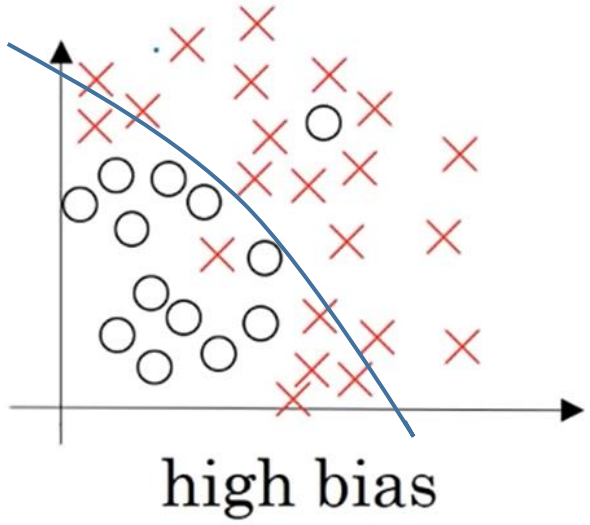
Era of small data: 70% data for training

Era of big data: 99% data for training

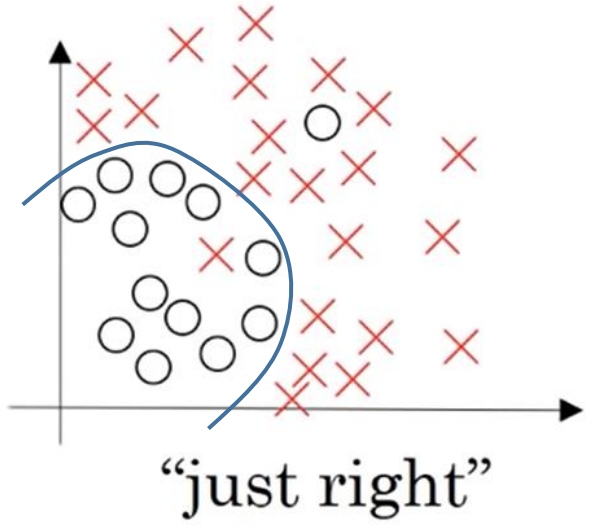
Make sure training set and test set come from the same distribution

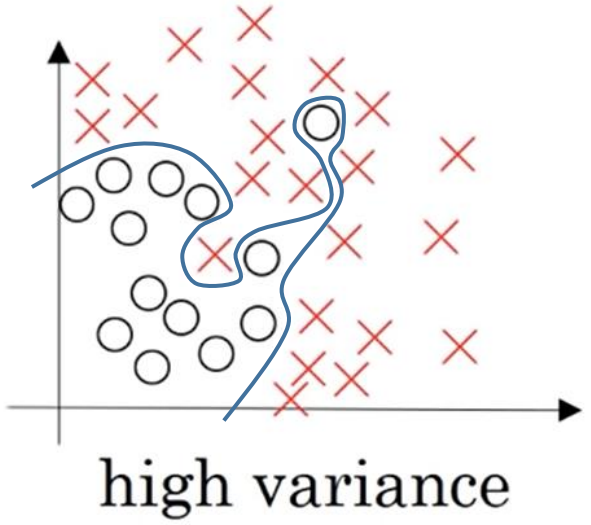
Not having a test set might be OK. (Only dev set)

Bias and Variance (Error)



Underfitting





Overfitting

Bias and Variance

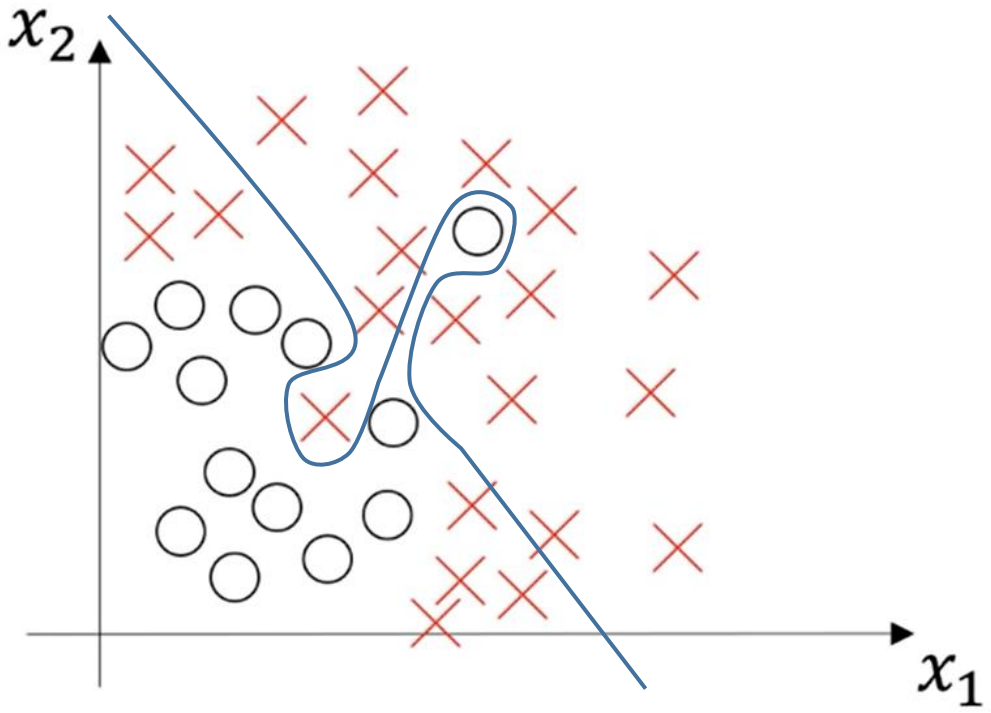
Cat classification

$y = 1$ $y = 0$

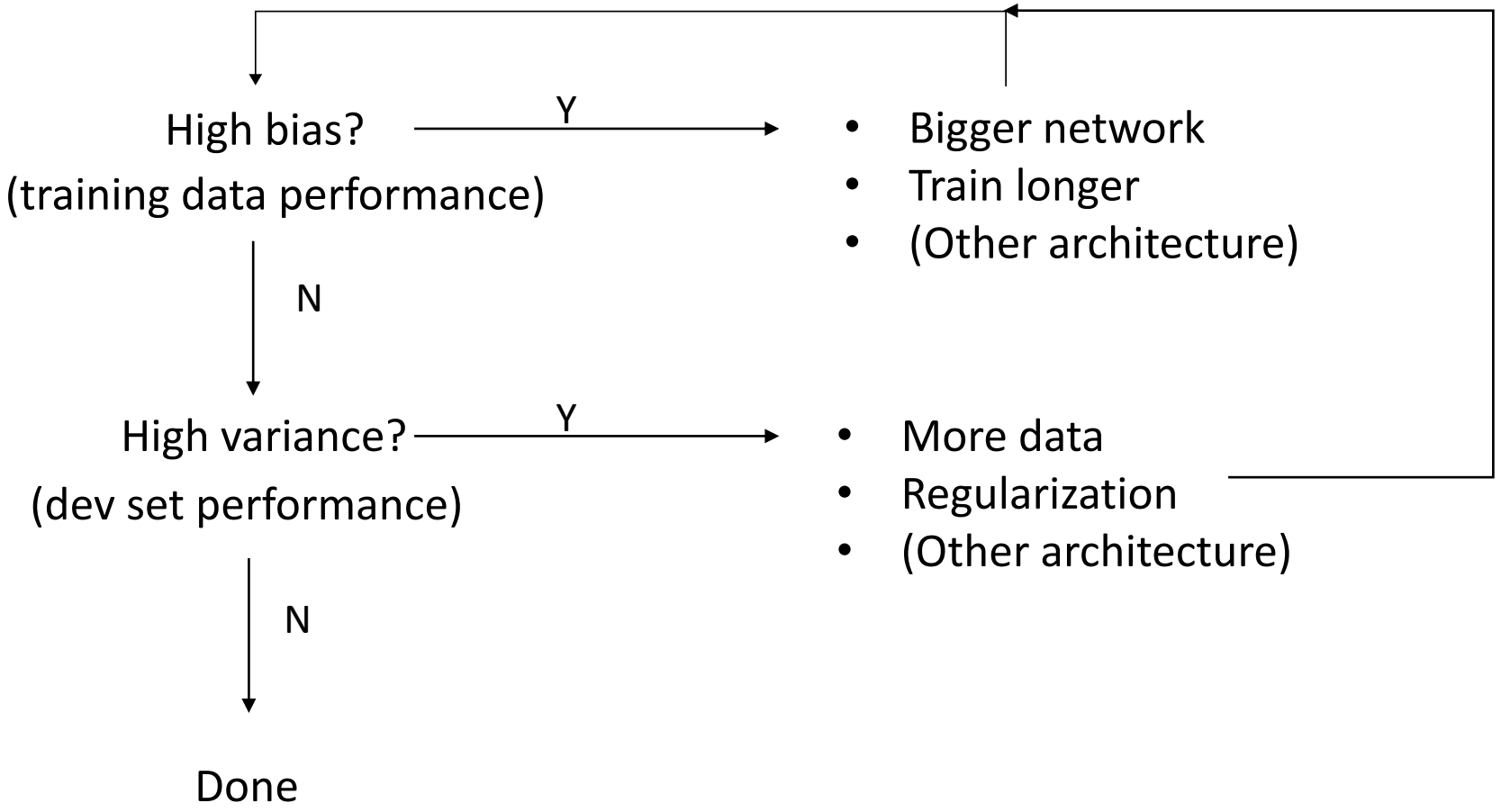

Human error $\approx 0\%$

Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high bias & high var	low bias & low var

High bias and high variance



Basic recipe for machine learning



Regularization

Regularization for logistic regression

$\min_{w,b} J(w,b)$ $\quad w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$ $\quad \lambda$: regularization parameter

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

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}||w||_1 \quad + \cancel{\frac{\lambda}{2m} b^2}$$

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

L1 regularization: $||w||_1 = \sum_{j=1}^{n_x} |w_j|$

After regularization, w will be sparse (i.e. having many zero elements)

Regularization for Neural network

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

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

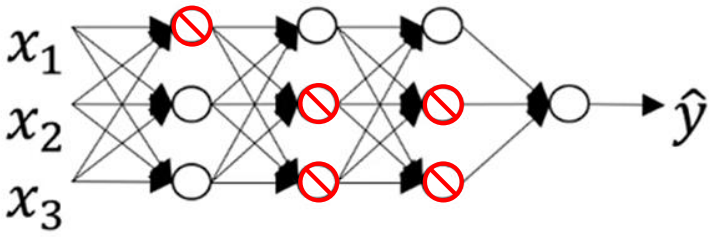
Frobenius norm, denoted as $||w^{[l]}||_F^2$

$$dw^{[l]} = dw_{before}^{[l]} + \frac{\lambda}{m} w^{[l]}$$

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

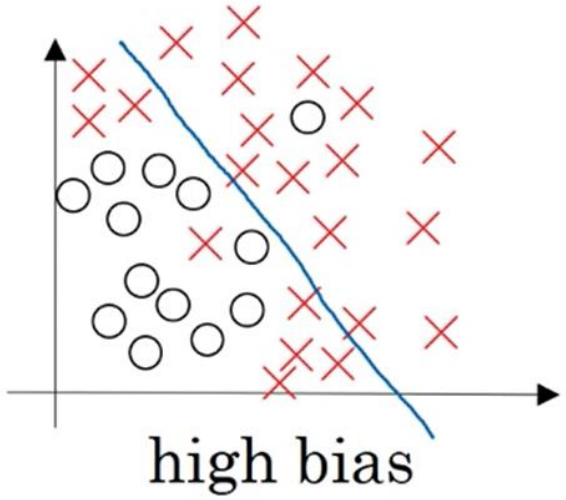
$$\begin{aligned} w^{[l]} &:= w^{[l]} - \alpha \cdot dw^{[l]} \\ &= w^{[l]} - \alpha \cdot (dw_{before}^{[l]} + \frac{\lambda}{m} w^{[l]}) \\ &= (\underbrace{1 - \alpha\frac{\lambda}{m}}_{< 1}) w^{[l]} - \alpha \cdot dw_{before}^{[l]} \end{aligned}$$

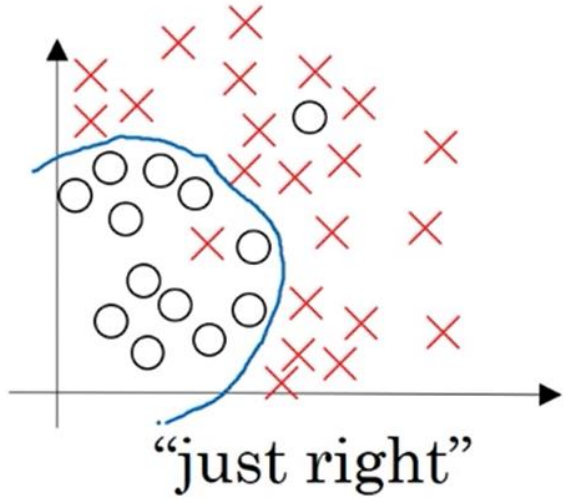
How does regularization prevent overfitting?

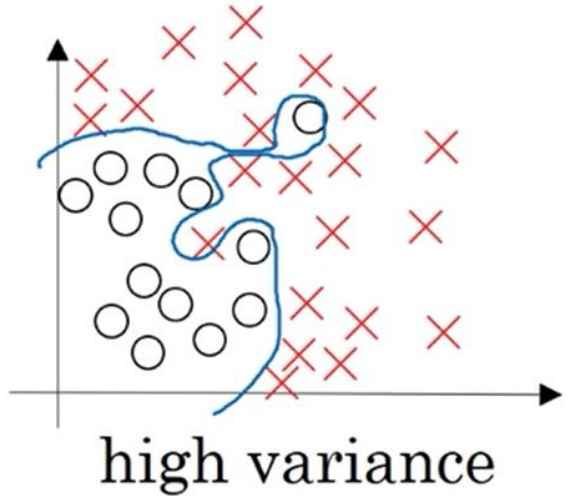


$$J(w^{[1]}, b^{[1]}, \cdots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} ||w^{[l]}||_F^2$$

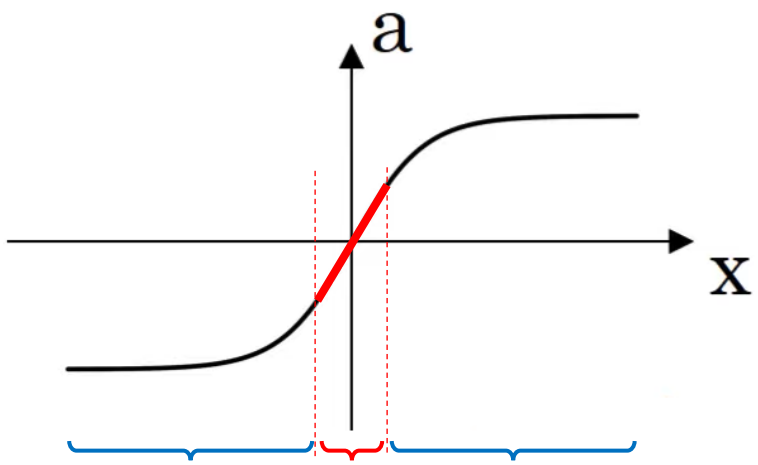
$\rightarrow w^{[l]} \approx 0$







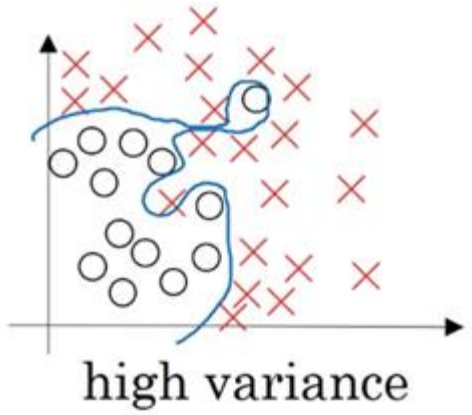
How does regularization prevent overfitting?



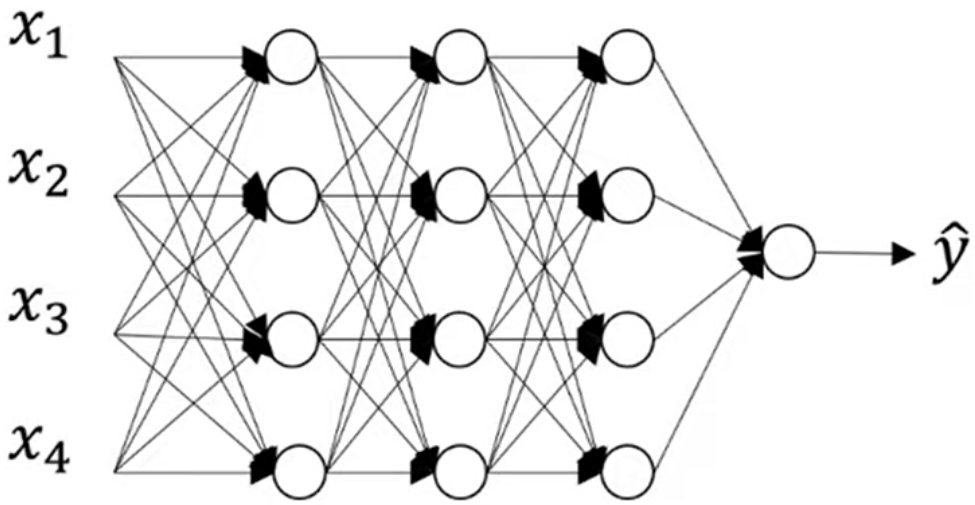
$g(z)$: $\tanh(z)$

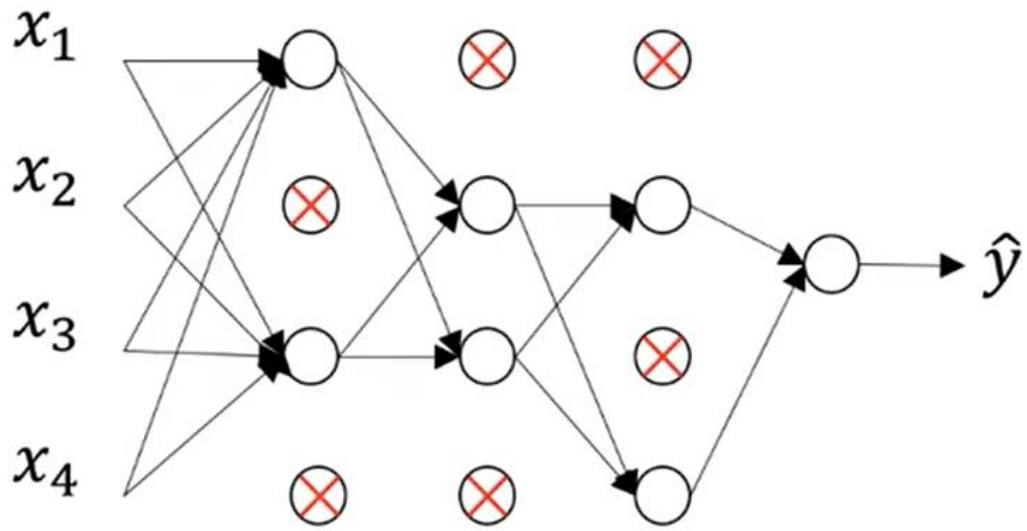
$\lambda \uparrow \qquad w^{[l]} \downarrow \qquad z^{[l]} = w^{[l]} a^{[l-1]} + b$

Every layer ≈ linear



Dropout regularization





Dropout probability = 0.5

Implementing dropout ("Inverted dropout")

Illustrate with layer $l = 3$

d3 = np.random.rand(a3.shape[0],a3.shape[1]) < keep_prob (keep_prob = 0.8)

a3 = np.multiply(a3,d3)

100 units ⟶ 20 units shut off

$z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$

↓

reduce by 20%

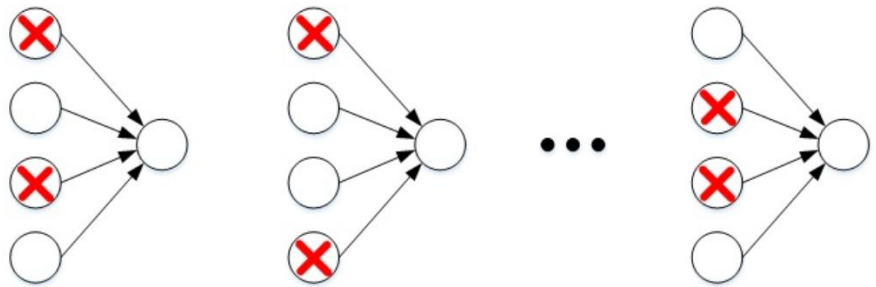
a3 /= keep_prob

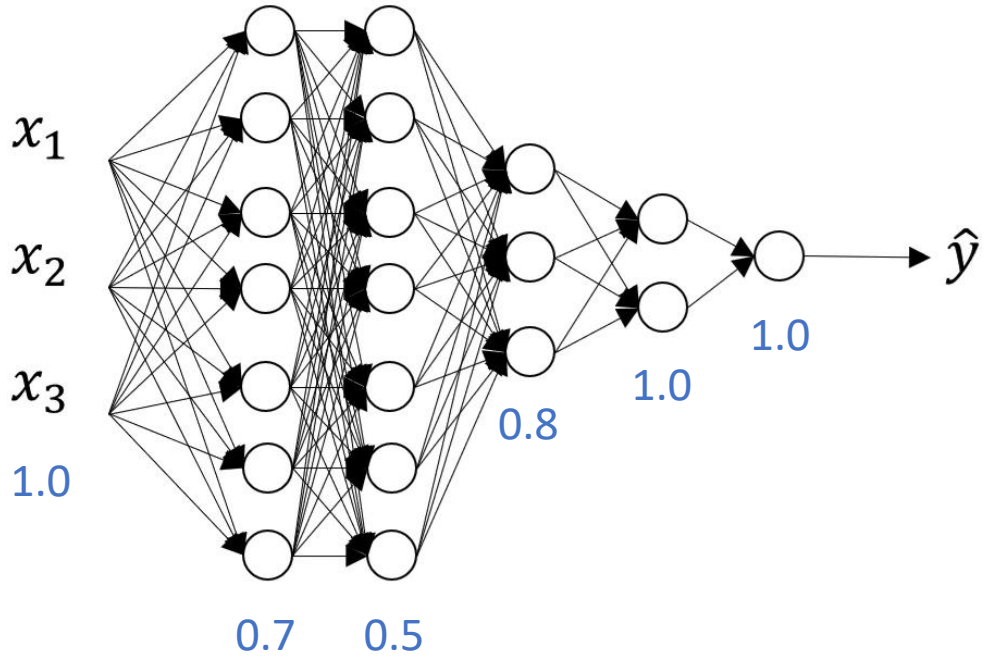
Dividing the a3 by keep_prob ensure the expected value of a3 remain the same.

Note: no dropout in test time

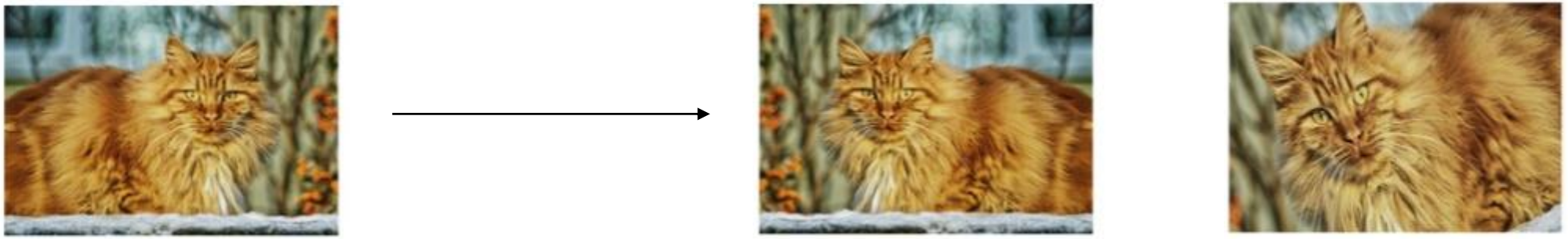
Why does drop-out work?

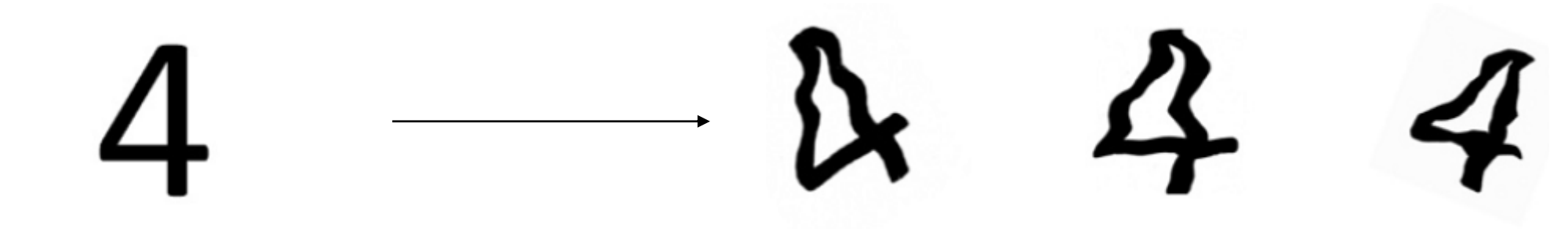
Intuition: Can't rely on any one feature, so have to spread out weights.





Other regularization methods (data augmentation)





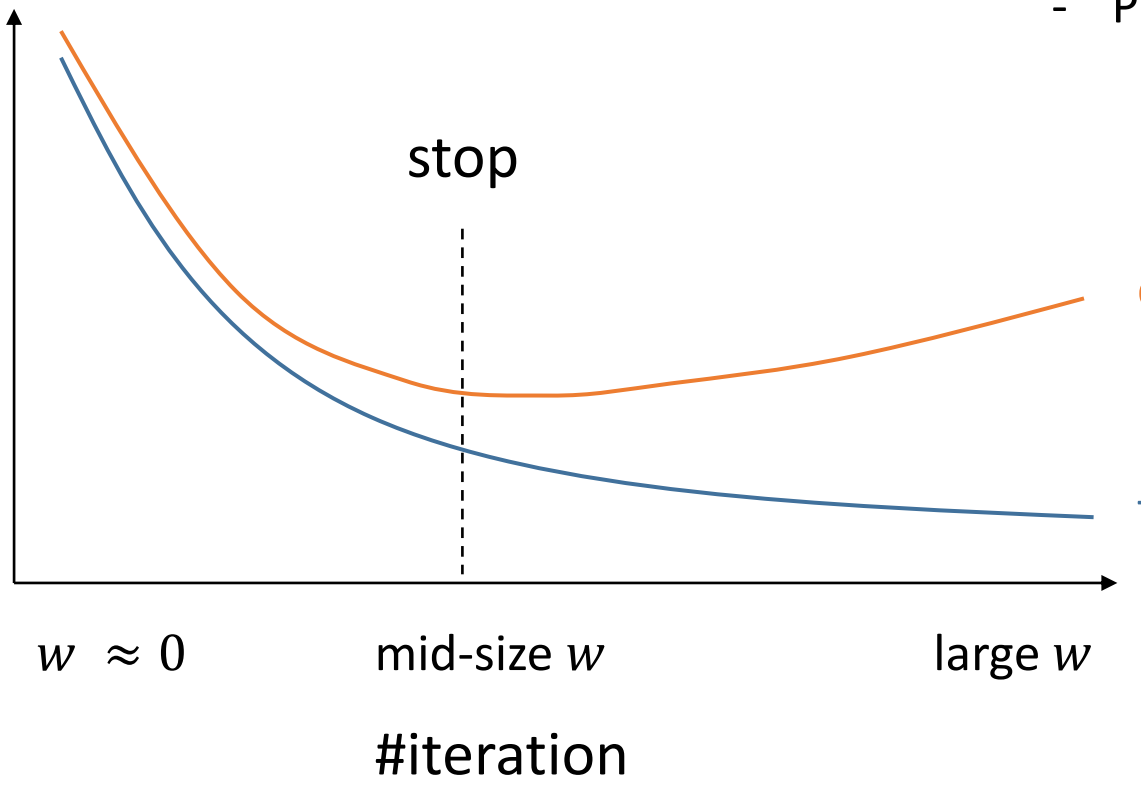
Other regularization methods (early stopping)

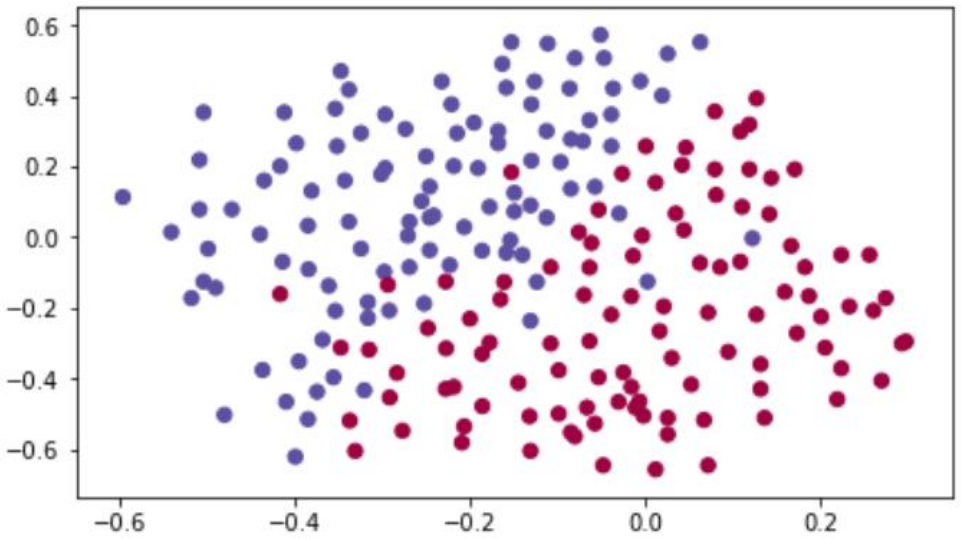
- Optimize cost function J

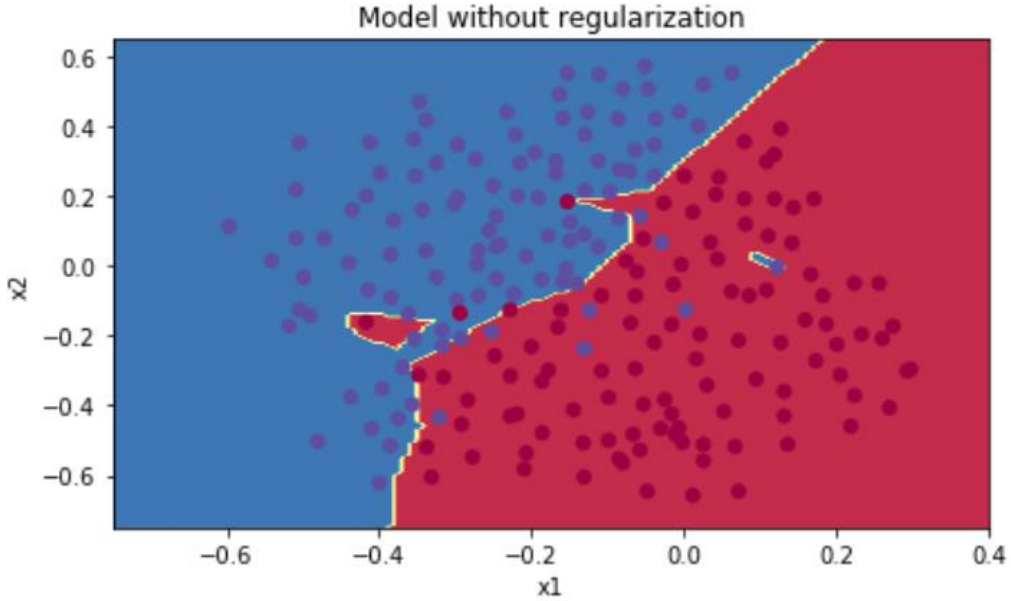
  Gradient descent, momentum,…

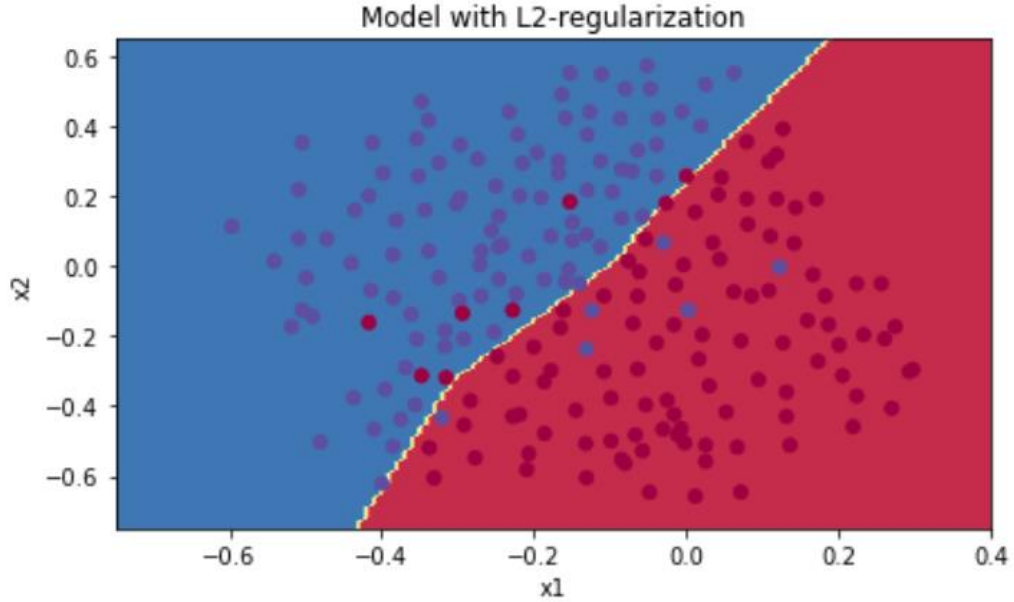
- Prevent overfitting

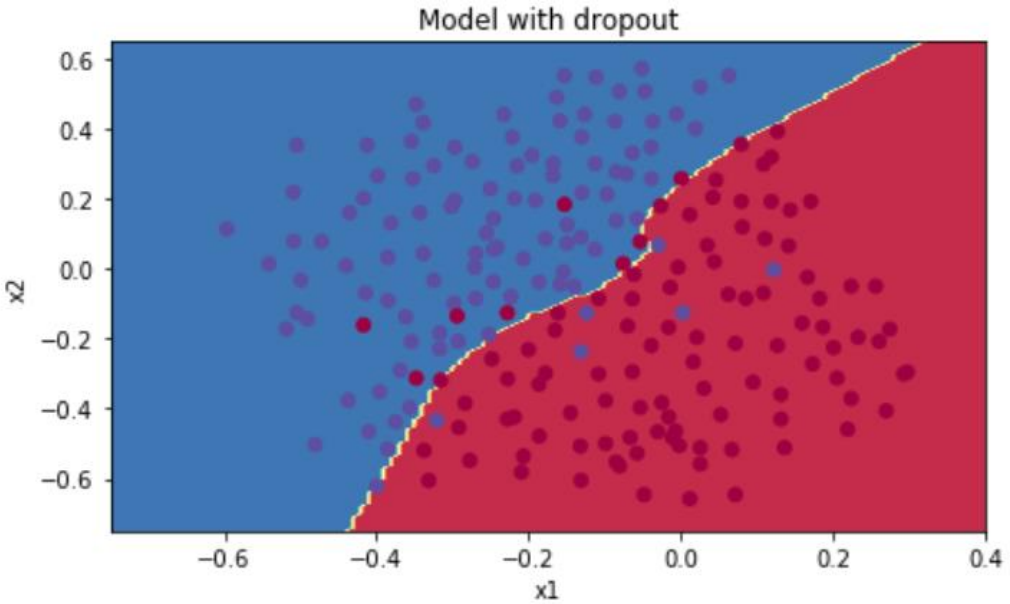
  Regularization, more data,…




0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.6
−0.4
−0.2
0.0
0.2


Model without regularization
x2
x1
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.6
−0.4
−0.2
0.0
0.2
0.4


Model with L2-regularization
x2
x1
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.6
−0.4
−0.2
0.0
0.2
0.4


Model with dropout
x2
x1
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.6
−0.4
−0.2
0.0
0.2
0.4

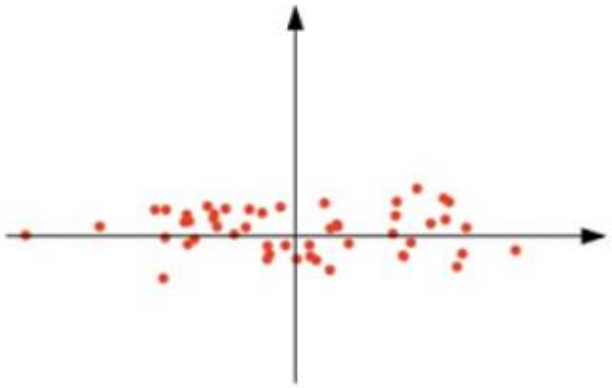
Setting up your optimization

Normalizing training set

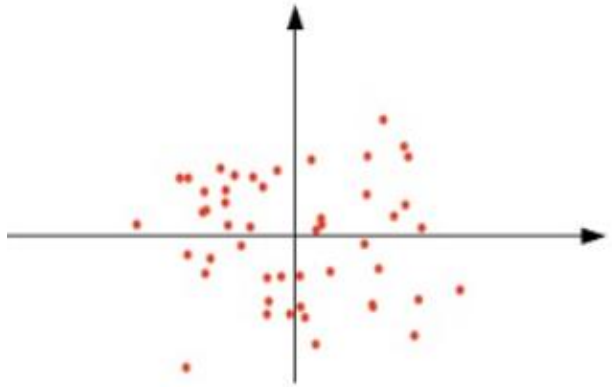
$$\mu = \frac{1}{m}\sum_{i=1}^{m} X^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (X^{(i)})^2$$

X



$X - \mu$



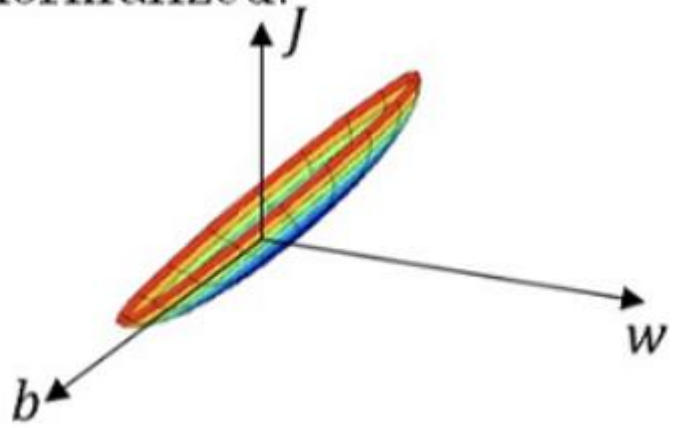
$$X := \frac{X - \mu}{\sigma^2}$$

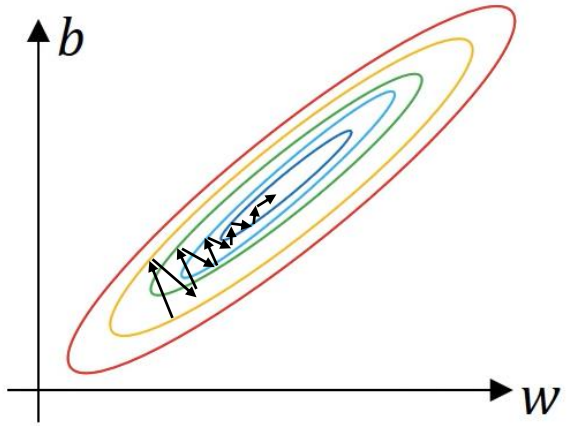
Note: use the same μ and σ for normalizing test set

Why normalize input?

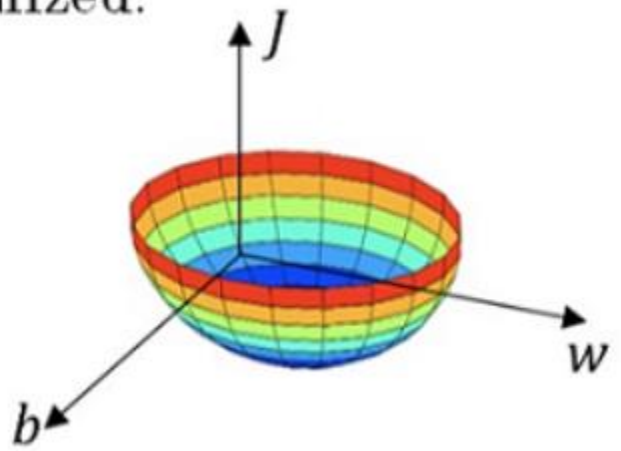
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

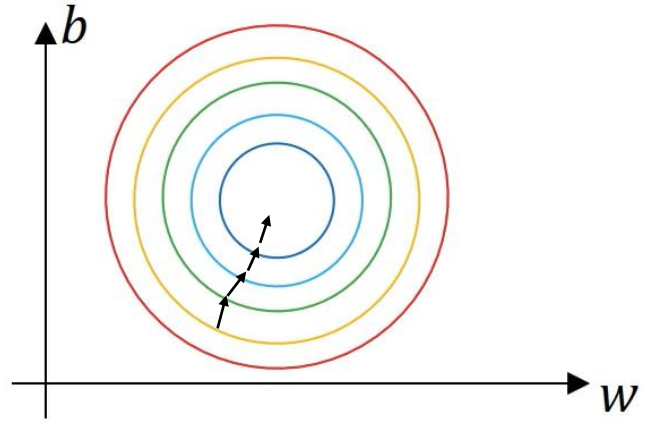
Unnormalized:



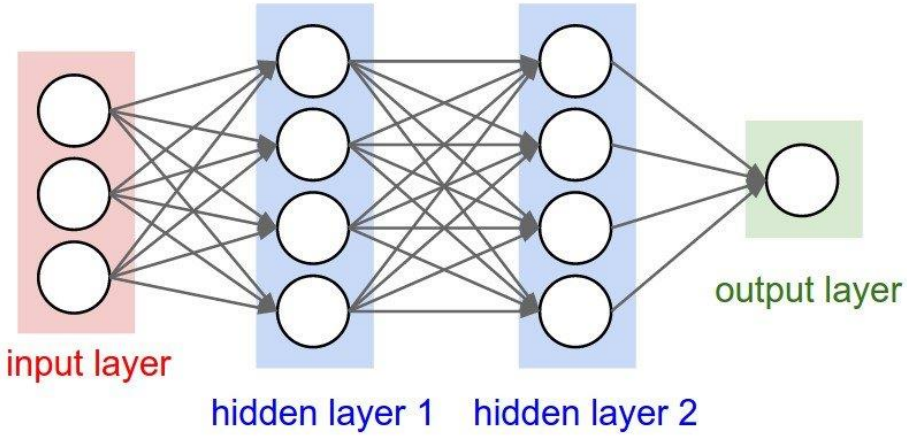


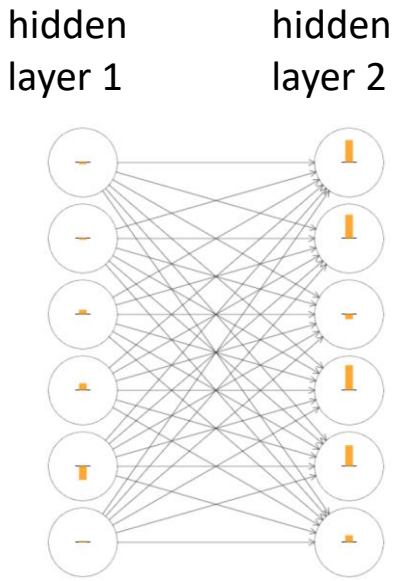
Normalized:



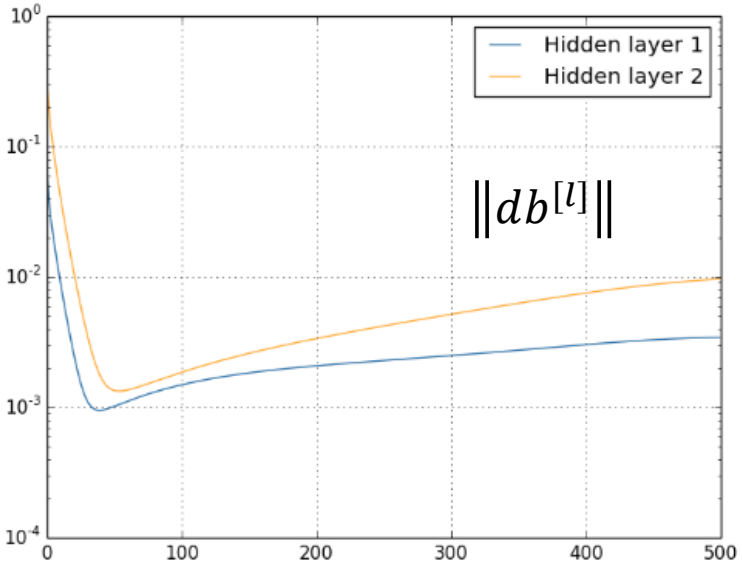


Vanishing/exploding gradient

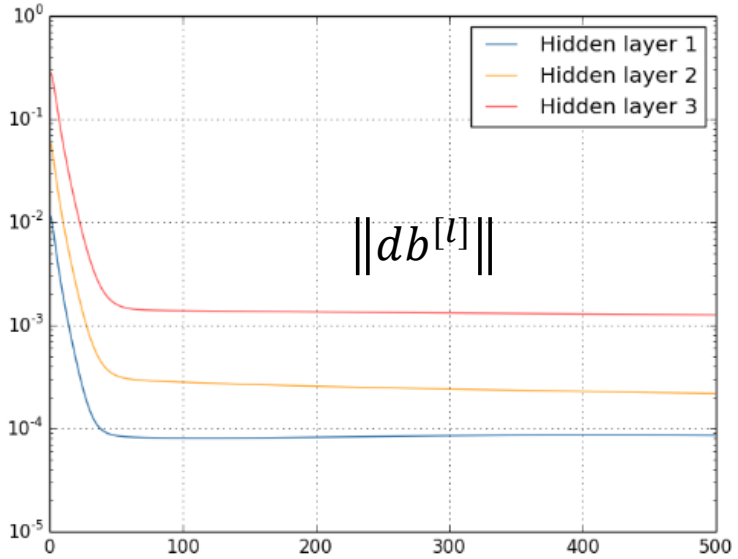




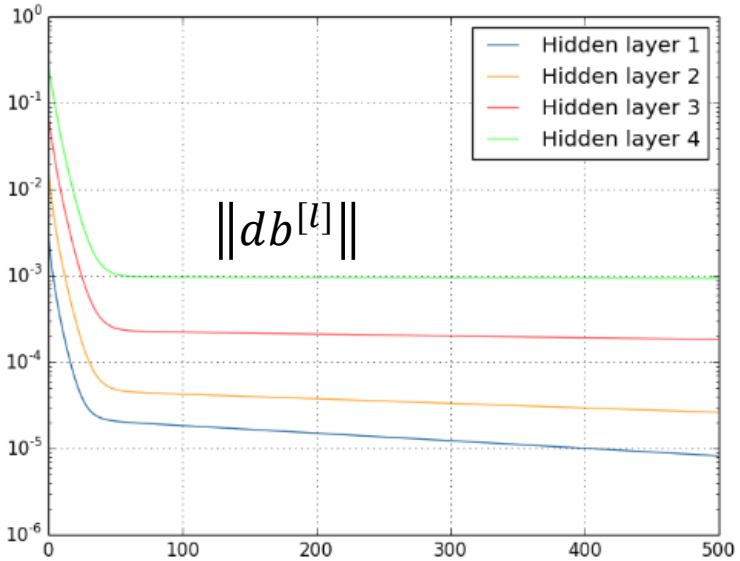
db in each nodes after initilization



number of iteration

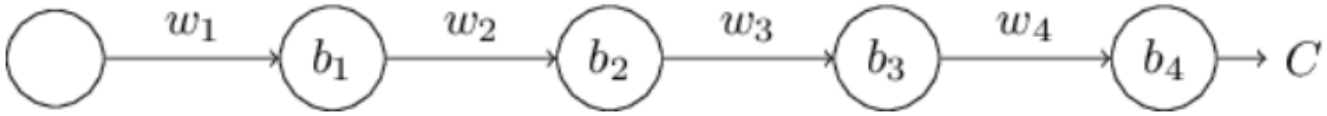


number of iteration

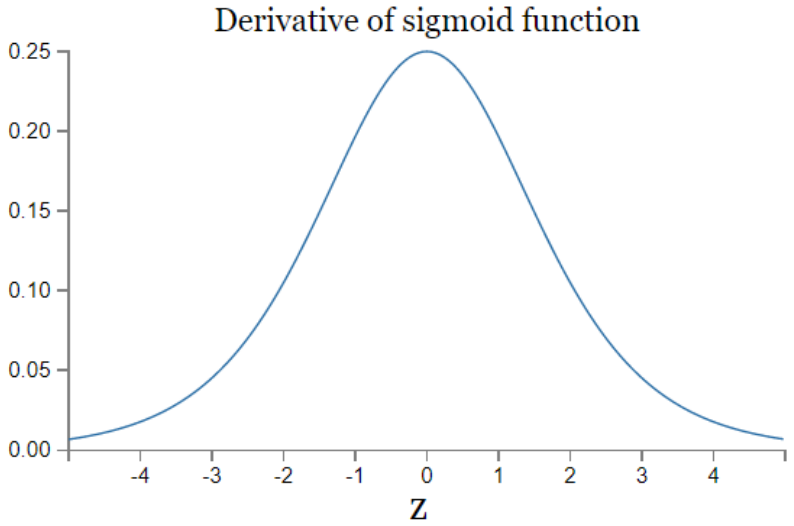


number of iteration

What causes vanishing/exploding gradient?



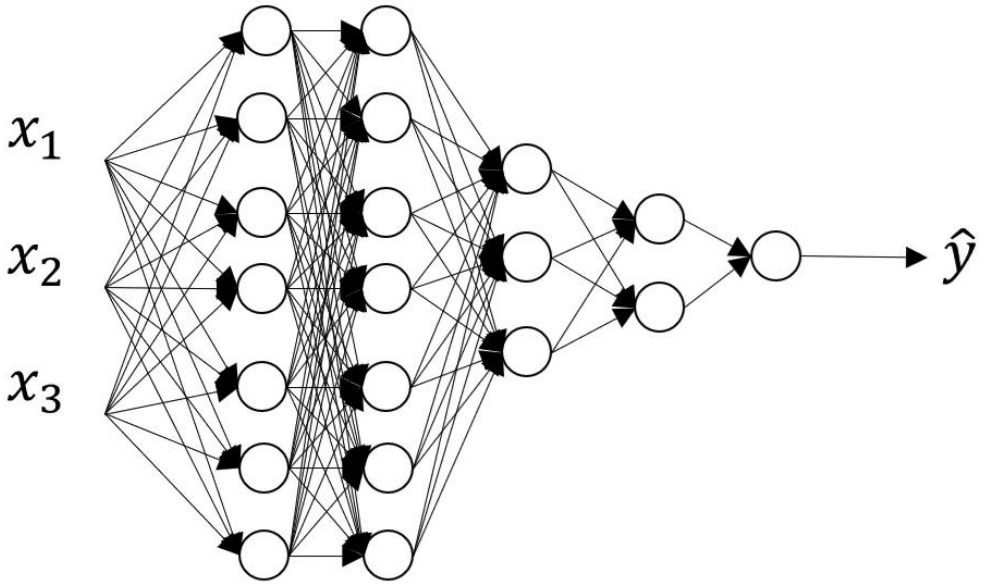
$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



$\sigma(z) \leq 1/4$ and $w^{[l]} \leq 1 \rightarrow w^{[l]}\sigma'(z^{[l]}) \leq 1/4$ $\rightarrow$ vanishing gradient in early layers

If $w^{[l]}$ is big, $w^{[l]}\sigma'(z^{[l]}) > 1$ $\rightarrow$ exploding gradient in early layers $\rightarrow$ large update (unstable network)

Weight Initialization in a Deep Network



$$dz^{[l]} = W^{[l+1]T} \cdot dz^{[l+1]} * g^{[l]'}(z^{[l]})$$

larger $n^{[l]} \rightarrow$ smaller w_{ij}

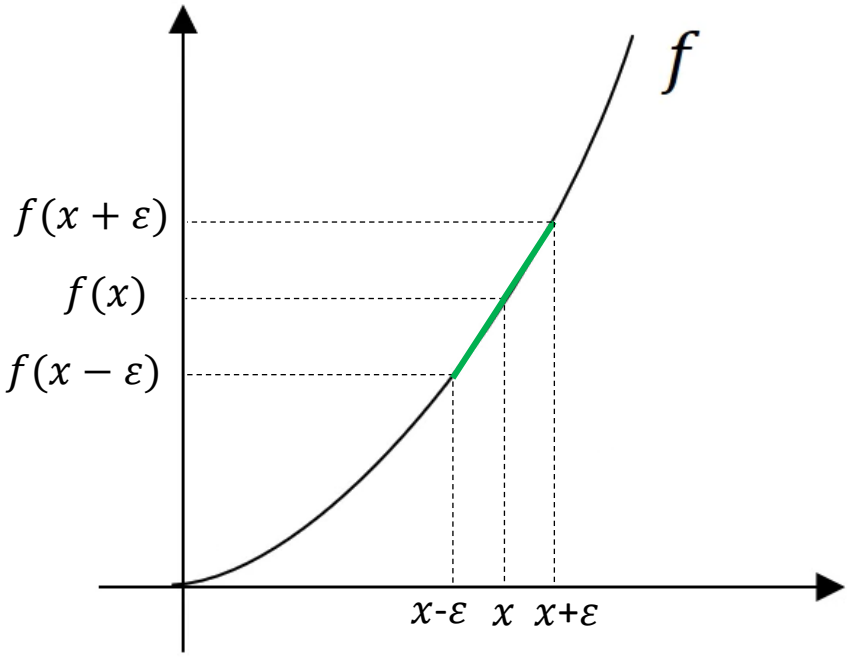
$$var(w_{ij}) = \frac{1}{n}$$

w [l] = np . random . randn ($n^{[l]}, n^{[l-1]}$) * np . sqrt (1 / $n^{[l-1]}$) tanh activation

w [l] = np . random . randn ($n^{[l]}, n^{[l-1]}$) * np . sqrt (2 / $n^{[l-1]}$) RELU activation

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

Numerical Approximations of Gradients



$$f'(x) \approx \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$ → High error $O(\varepsilon)$

$$f'(x) \approx \frac{f(x+\varepsilon) - f(x-\varepsilon)}{\varepsilon}$$ → Low error $O(\varepsilon^2)$

Gradient check for a neural network

Take $W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ.

$$J\left(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}\right) = J(\theta)$$

Take $dW^{[1]}, db^{[1]}, \ldots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $\mathrm{d}\theta$.

Is $\mathrm{d}\theta$ the gradient of $J(\theta)$?

Gradient check

$$J\left(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}\right) = J(\theta) = J(\theta_1, \theta_2, \dots, \theta_i, \dots)$$

For each i:

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \cdots, \theta_i + \varepsilon, \cdots) - J(\theta_1, \theta_2, \cdots, \theta_i - \varepsilon, \cdots)}{2\varepsilon} \approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$$

Check: $$\frac{||d\theta_{approx} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2} > 10^{-3} \rightarrow \text{bug}$$

$\varepsilon = 10^{-7}$

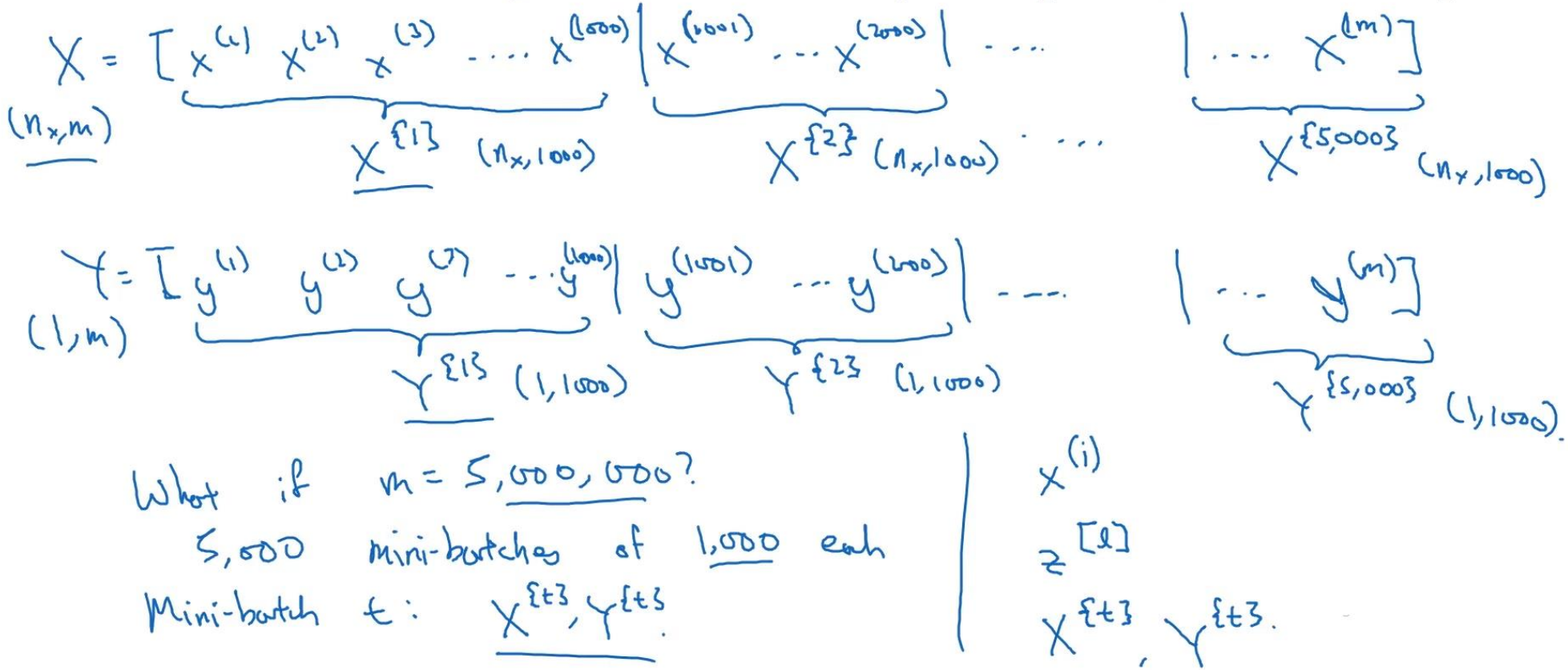
Note:

- Don't check in training (only to debug)
- Remember if cost function has regularization
- Doesn't work with dropout

Optimization algorithms

Batch vs. mini-batch gradient descent

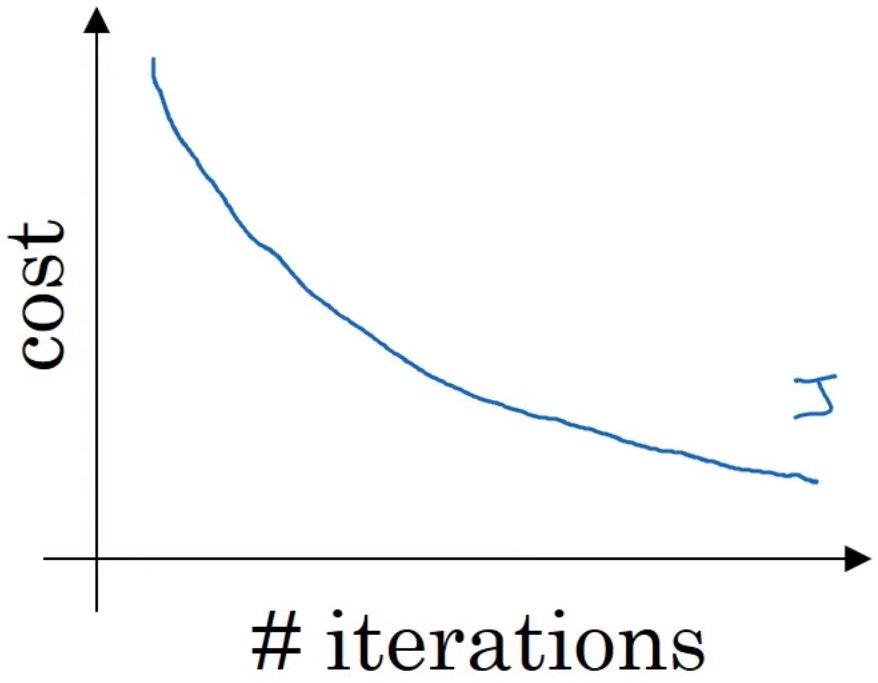
Vectorization allows you to efficiently compute on m examples.



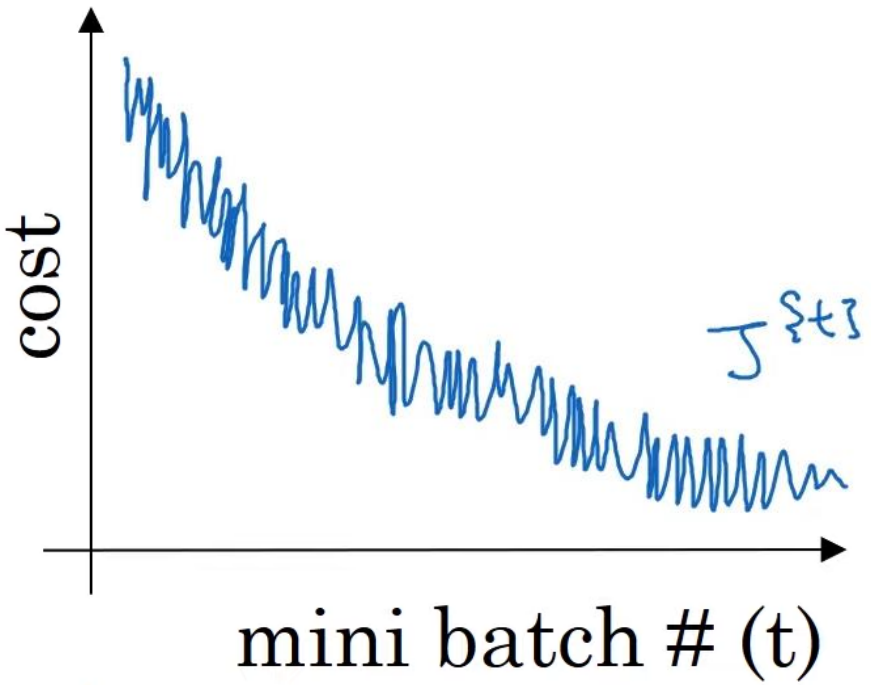
Epoch: one pass through training set (with include T mini-Batches)

Training with mini batch gradient descent

Batch gradient descent

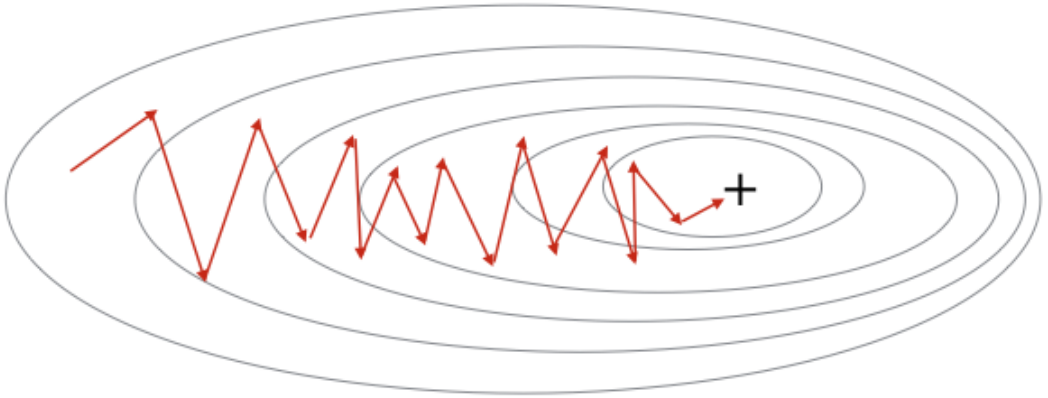


Mini-batch gradient descent

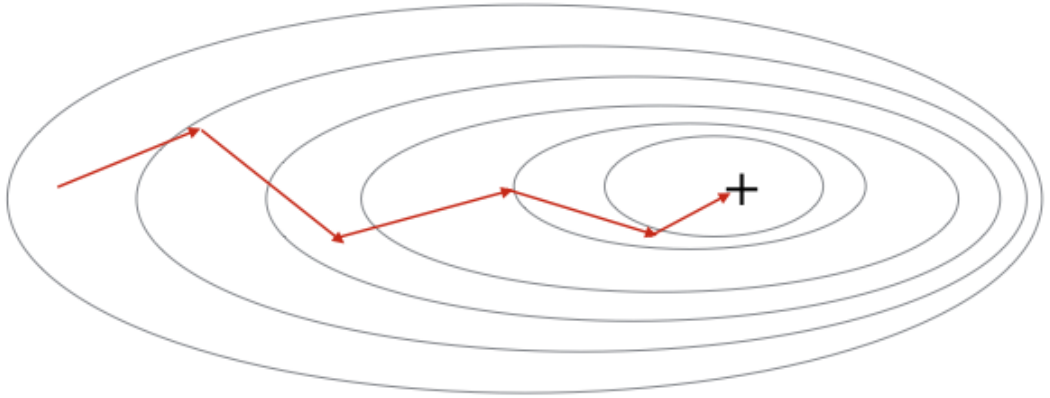


Stochastic vs Mini-Batch

Stochastic Gradient Descent



Mini-Batch Gradient Descent

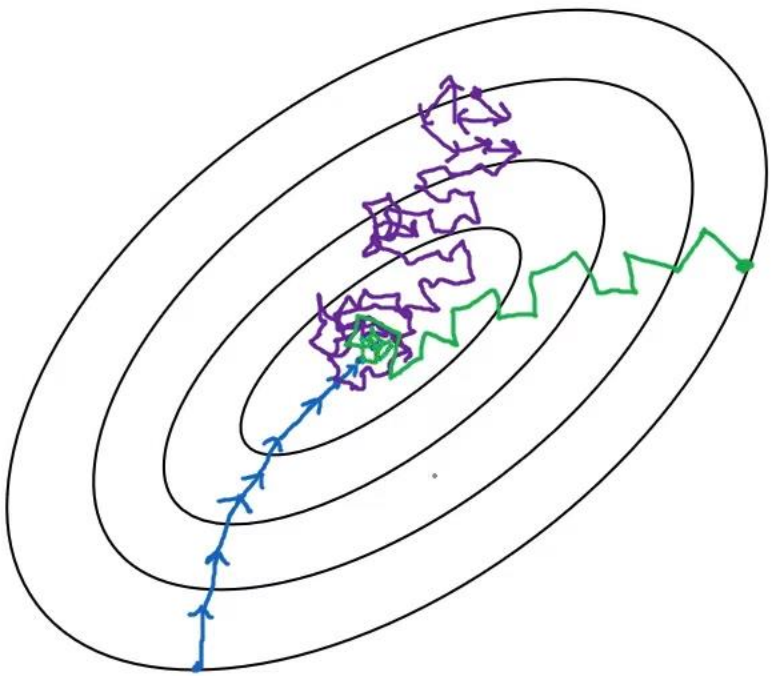


Choosing your mini-batch size

→ 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. $(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)}) \dots (x^{(2)}, y^{(1)})$

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 1000)
- Make progress without processing entire training set.

Batch gradient descent (mini-batch size = m)

↓

Too long per iteration

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}

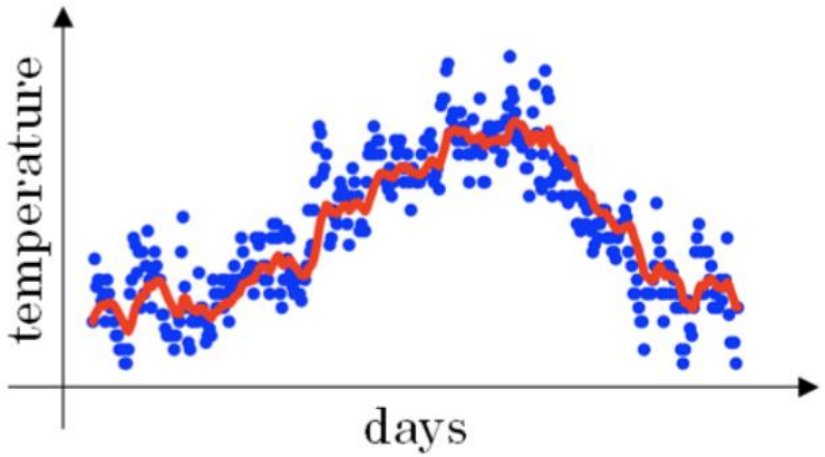
Make sure mini-batch fit in CPU/GPU memory.

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

Exponentially Weighted Averages

The temperature changes in London in six months:

$\theta_1 = 40°\text{F}$
$\theta_2 = 49°\text{F}$
$\theta_3 = 45°\text{F}$
$\vdots$
$\theta_{180} = 60°\text{F}$
$\theta_{181} = 56°\text{F}$
$\vdots$



$$V_0 = 0$$

$$V_1 = 0.9\, V_0 + 0.1\, \theta_1$$

$$\begin{aligned} V_2 &= 0.9\, V_1 + 0.1\, \theta_2 \\ &= 0.9\,(0.9\, V_0 + 0.1\, \theta_1) + 0.1\, \theta_2 \\ &= 0.9^2 V_0 + 0.9 \cdot 0.1\, \theta_1 + 0.1\, \theta_2 \end{aligned}$$

$$\begin{aligned} V_3 &= 0.9\, V_2 + 0.1\, \theta_3 \\ &= 0.9(0.9^2 V_0 + 0.9 \cdot 0.1\, \theta_1 + 0.1\, \theta_2) + 0.1\, \theta_3 \\ &= 0.9^3 V_0 + 0.9^2 \cdot 0.1\, \theta_1 + 0.9 \cdot 0.1\, \theta_2 + 0.1\, \theta_3 \end{aligned}$$

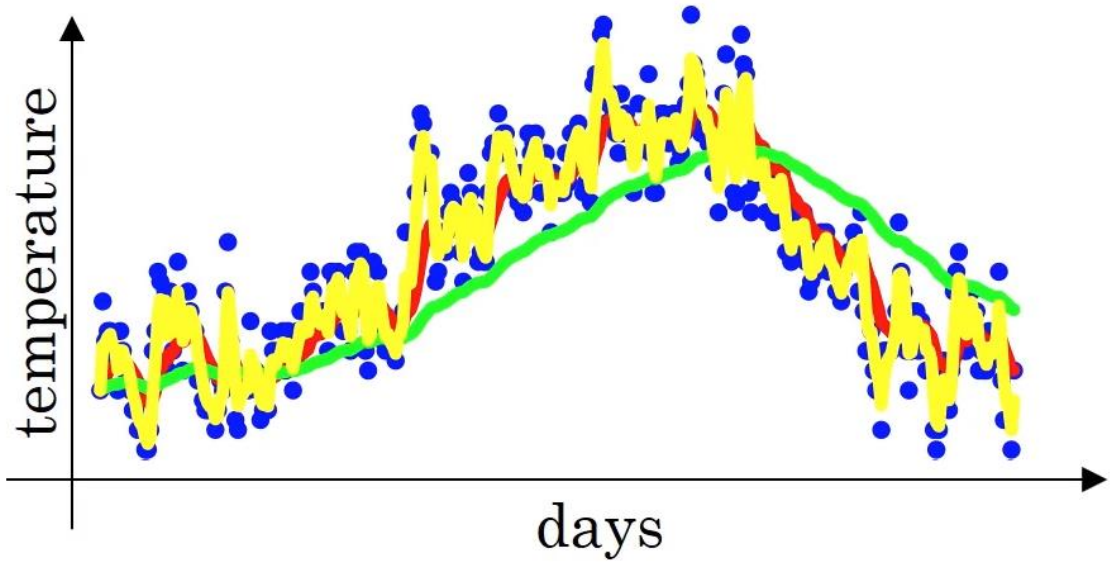
$$\begin{aligned} V_t &= 0.9\, V_{t-1} + 0.1\, \theta_t \\ &= 0.9^t V_0 + 0.9^{t-1} \cdot 0.1\, \theta_1 + 0.9^{t-2} \cdot 0.1\, \theta_2 + \cdots + 0.9 \cdot 0.1\, \theta_{t-1} + 0.1\, \theta_t \end{aligned}$$

Exponentially weighted averages

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

V_t is approximately the average over $\approx \frac{1}{1-\beta}$ days

$\beta = 0.9$: $\approx$ 10 days temperature
$\beta = 0.98$: $\approx$ 50 days
$\beta = 0.5$: $\approx$ 2 days

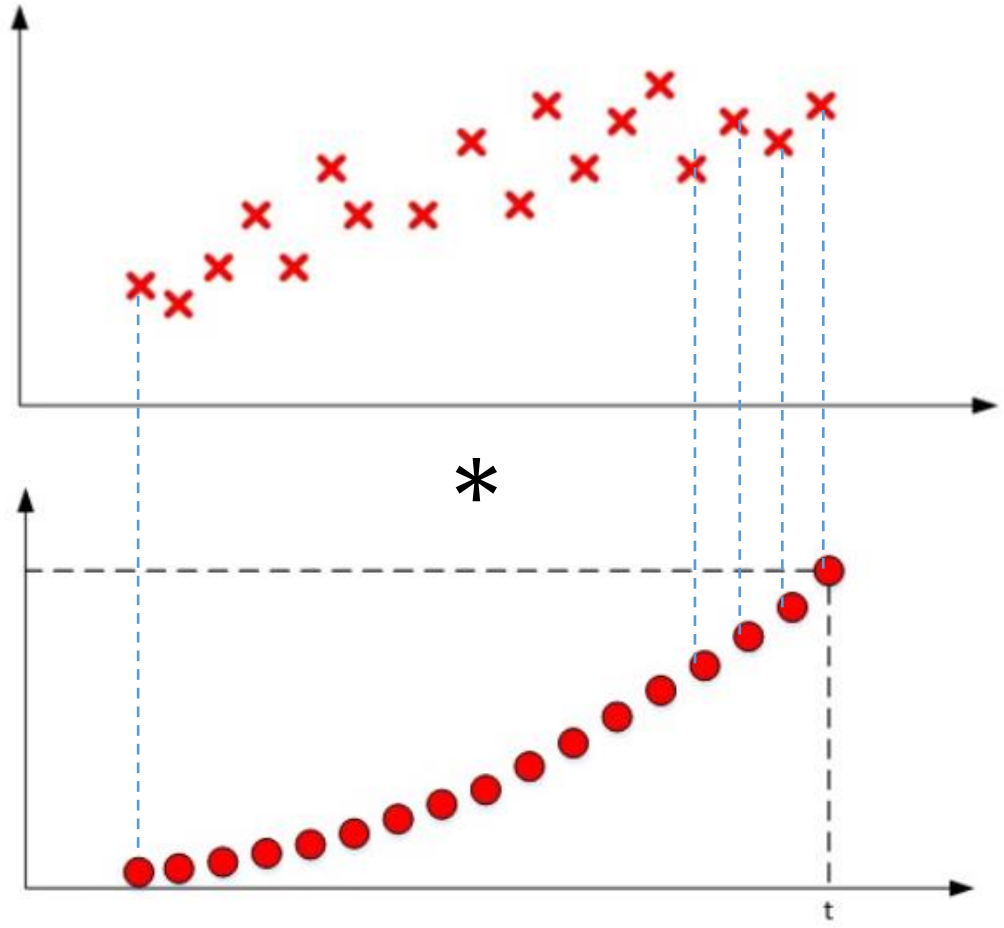


Exponentially weighted averages

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

$$v_{100} = 0.9v_{99} + 0.1\theta_{100}$$
$$v_{99} = 0.9v_{98} + 0.1\theta_{99}$$
$$v_{98} = 0.9v_{97} + 0.1\theta_{98}$$

$$\begin{aligned} V_{100} = \quad & 0.1 \times 0.9^0 \times \theta_{100} \\ & +0.1 \times 0.9^1 \times \theta_{99} \\ & +0.1 \times 0.9^2 \times \theta_{98} \\ & +0.1 \times 0.9^3 \times \theta_{97} \\ & +0.1 \times 0.9^4 \times \theta_{96} \\ & + \cdots \end{aligned}$$



Implementing exponentially weighted averages

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

$$V_\theta = 0$$

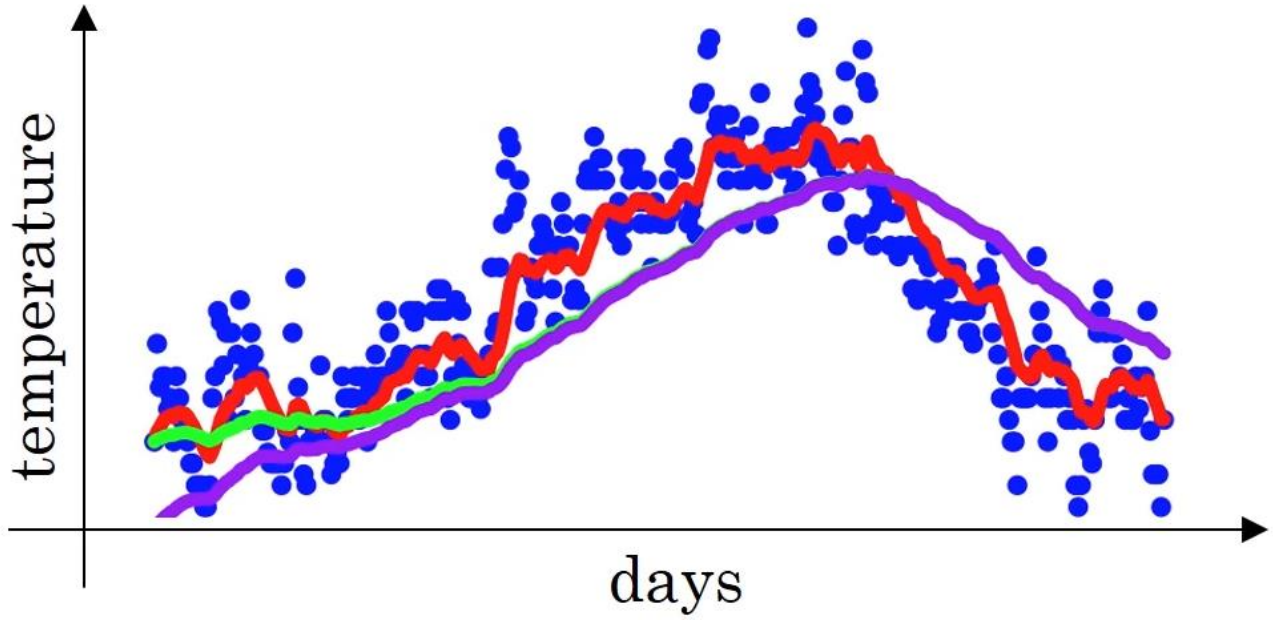
$$Repeat\ \{$$

$$Get\ next\ \theta_t$$

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

$$\}$$

Bias correction



$\beta = 0.9$

$\beta = 0.98$

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

$V_0 = 0$

$V_1 = \quad 0 \quad +0.02 \times \theta_1$

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

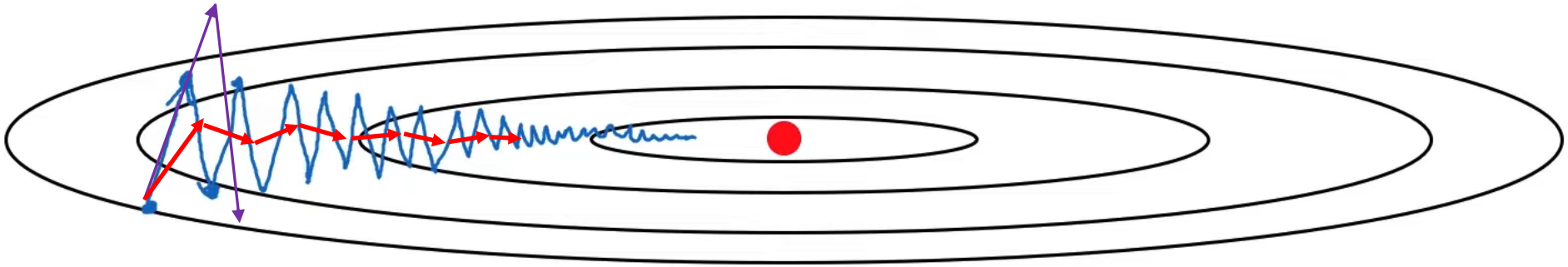
$= 0.0196 \times \theta_1 + 0.02 \times \theta_2$

Bias correction $\quad v_{t_{correct}} := \dfrac{v_t}{1-\beta^t}$

$V_{1_{correct}} = \dfrac{0.02 \times \theta_1}{0.02} = \theta_1$

$V_{2_{correct}} = \dfrac{0.0196 \times \theta_1 + 0.02 \times \theta_2}{0.0396} = 0.495 \times \theta_1 + 0.505 \times \theta_2$

Gradient descent example



Momentum

On iteration t:

Compute dW, db on current mini-batch.

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

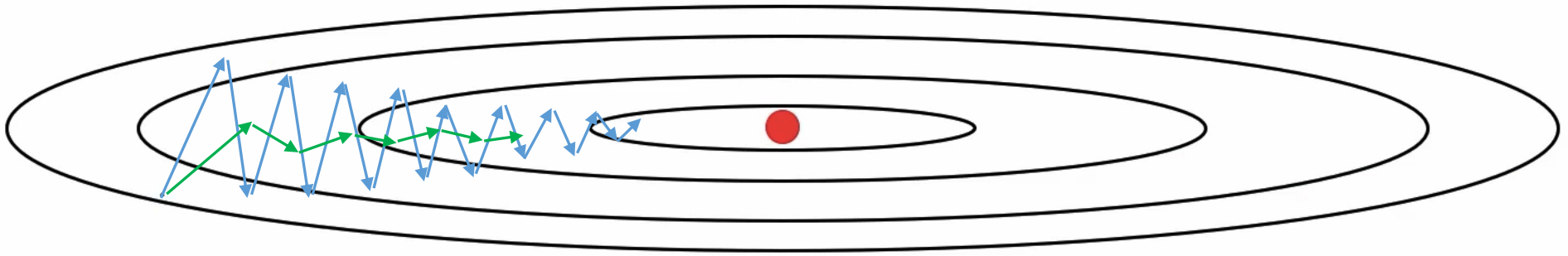
$V_{db} = \beta V_{db} + (1 - \beta) d_b$

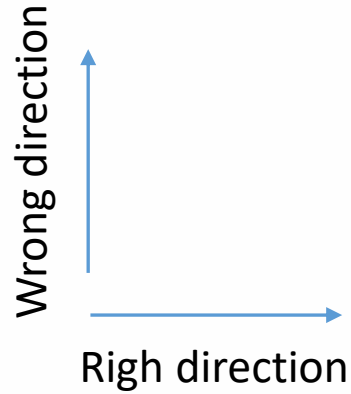
$W = W - \alpha V_{dW} \quad , \quad b = b - \alpha V_{db}$

Hyperparameter: α, β

$\beta = 0.9$ (average last 10 gradient)

RMSprop





We need to reduce the update in the wrong direction.

With many dimension, the projections on the wrong directions usually larger than that of right direction.

On iteration t

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

$$S_{db} = \beta S_{db} + (1-b)\, d\, b^2$$

$$W := W - a\frac{dW}{\sqrt{S_{dW}}},\ b := b - a\frac{db}{\sqrt{S_{db}}} \Rightarrow W := W - a\frac{dW}{\sqrt{S_{dW}} + e},\ b := b - a\frac{db}{\sqrt{S_{db}} + e}$$

$\neq 0$

$$e = 10^{-8}$$

Intuition: Slow down update on wrong direction

If vertical (wrong direction) are $w_1, w_4, \dots, b_3, b_8, \dots$

Then $\sqrt{S_{dW_1}}, \sqrt{S_{dW_4}}, \dots, \sqrt{S_{db_3}}, \sqrt{S_{db_8}}, \dots$ are usually larger

After reducing update in wrong direction, we can use larger α

Adam (Adaptive Moment Estimation) optimization algorithm

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

$On\ iteration\ t:$

Compute dW, db using current mini-batch

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

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

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

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

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

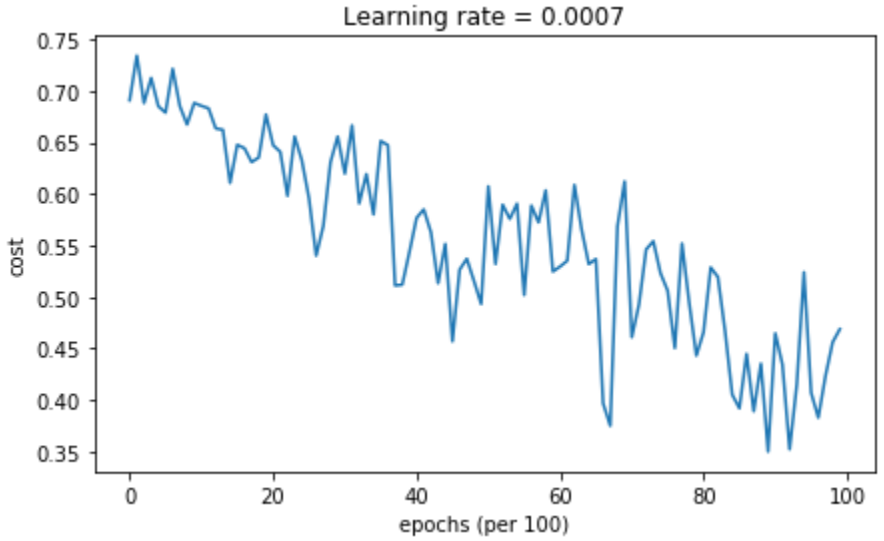
Hyperparameter: $\alpha, \beta_1, \beta_2, \varepsilon$

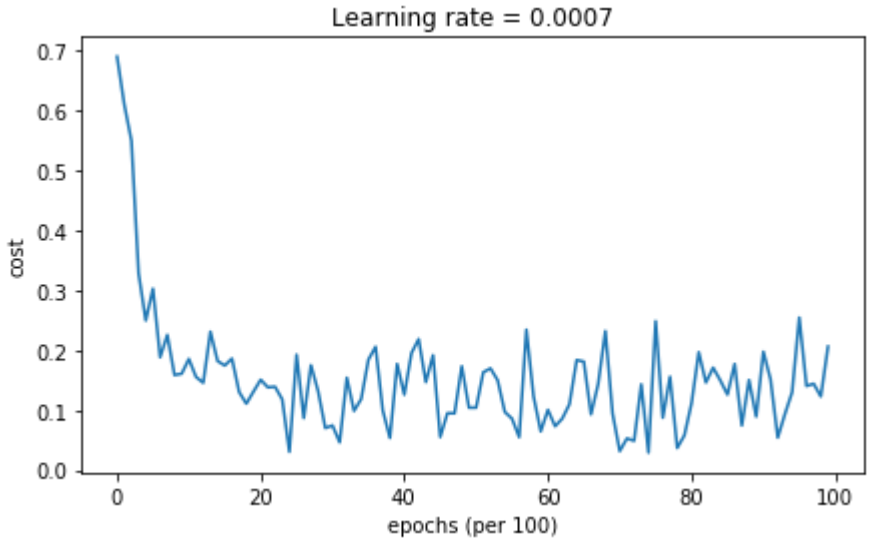
$\beta_1 = 0.9$

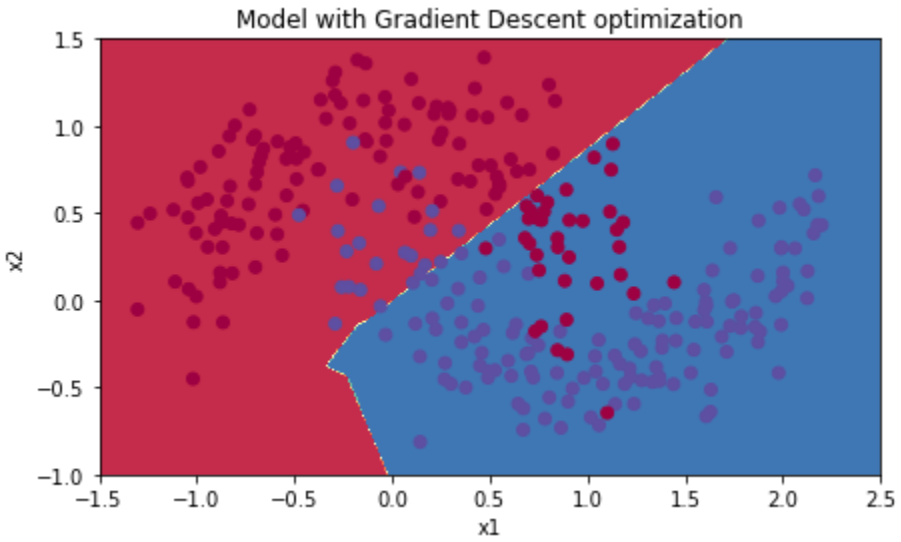
$\beta_2 = 0.999$

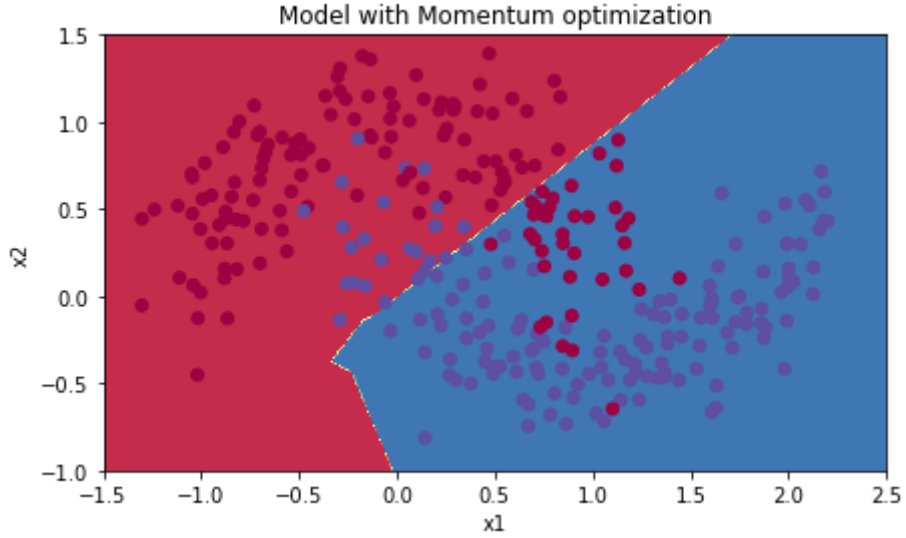
$\varepsilon = 10^{-8}$

Gradient descent vs Adam optimization



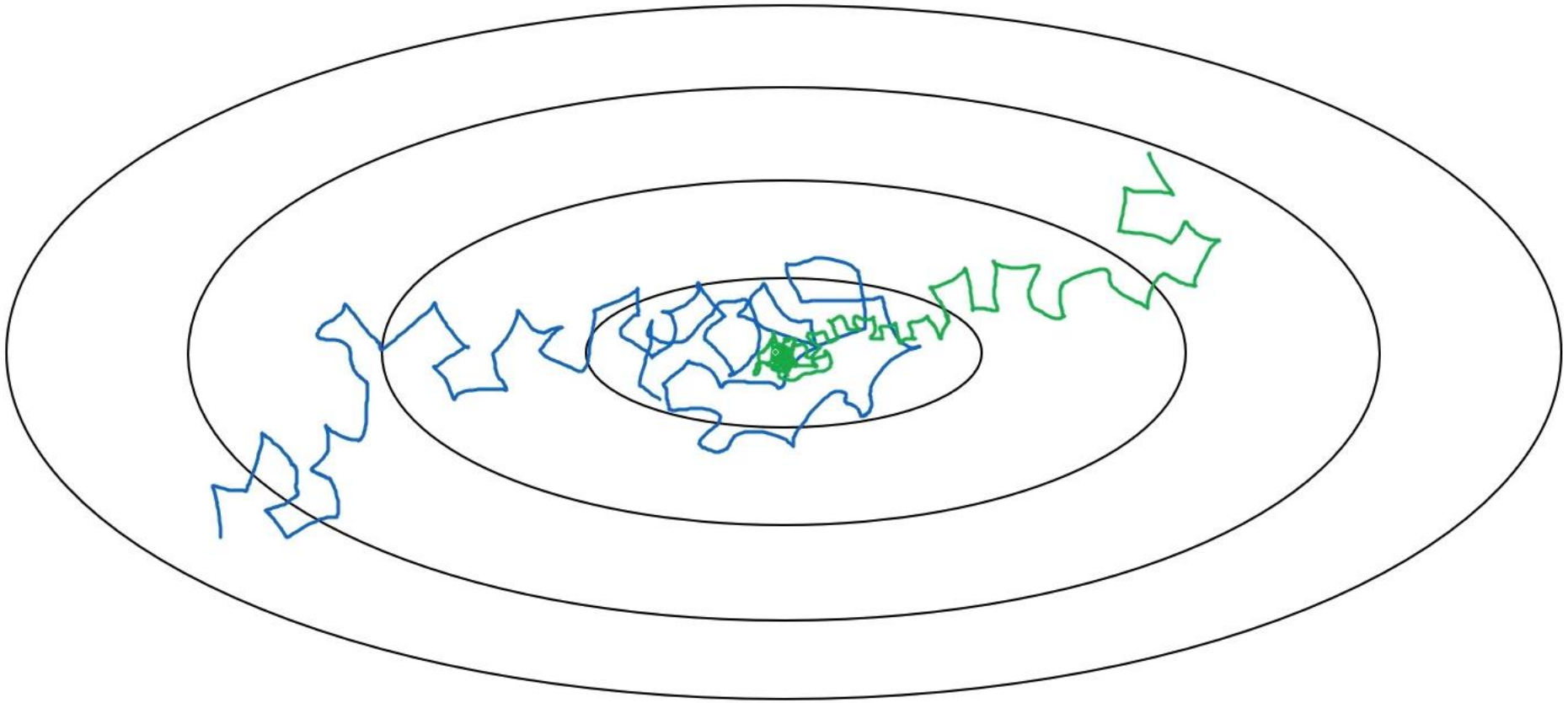






Learning rate decay

Slowly reduce α



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

$$\alpha = 0.95^{epoch} \cdot \alpha_0$$

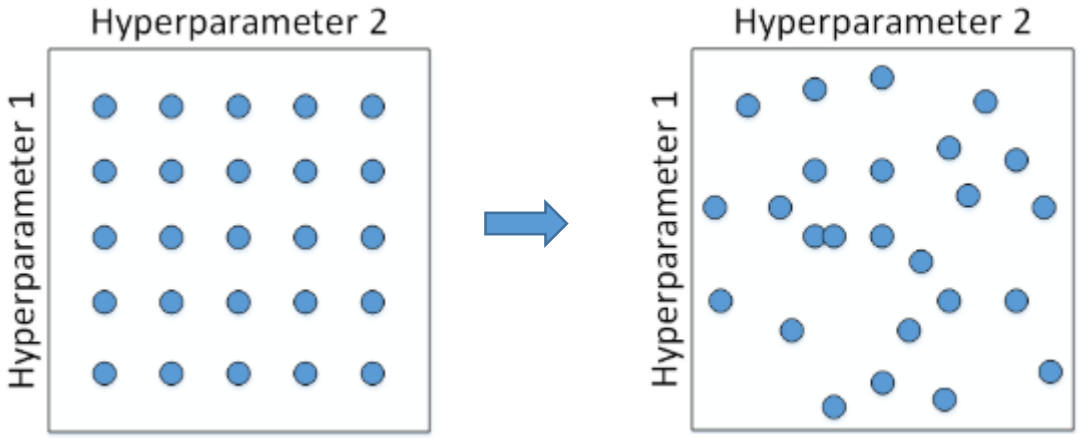
$$\alpha = \frac{k}{\sqrt{epoch}} \cdot \alpha_0 \quad or \quad \frac{k}{\sqrt{t}} \cdot \alpha_0$$

Hyperparameter tuning

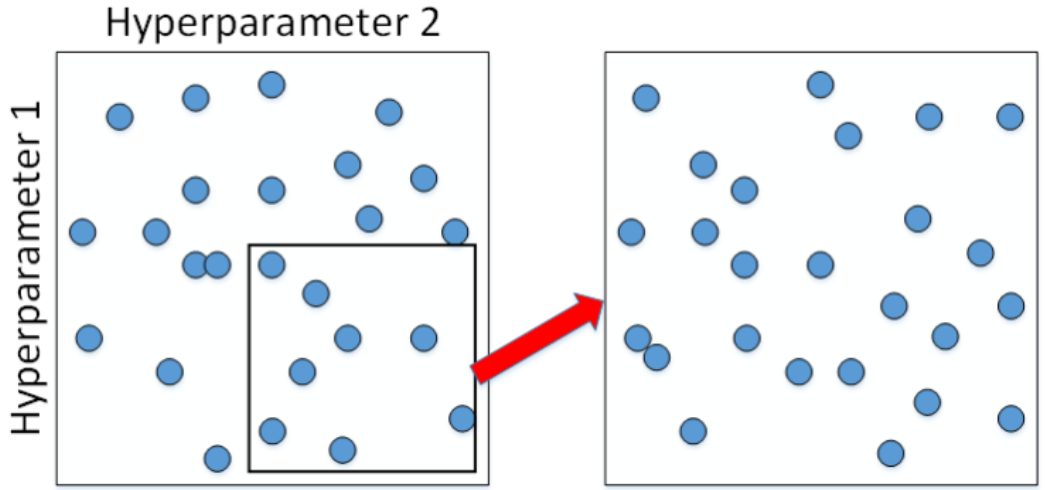
Tuning process

- α: Learning factor
- β: Momentum gradient descent factor 0.9
- $\beta_1, \beta_2, \varepsilon$: Adam algorithm parameters 0.9 , 0.999 , 10^{-8}
- #layers: Number of neural network layers
- #hidden units: the number of neurons in each hidden layer
- learning rate decay: learning factor decay parameter
- mini-batch size: the number of samples contained in the batch training sample

Searching scheme



Use random (don’t use grid)

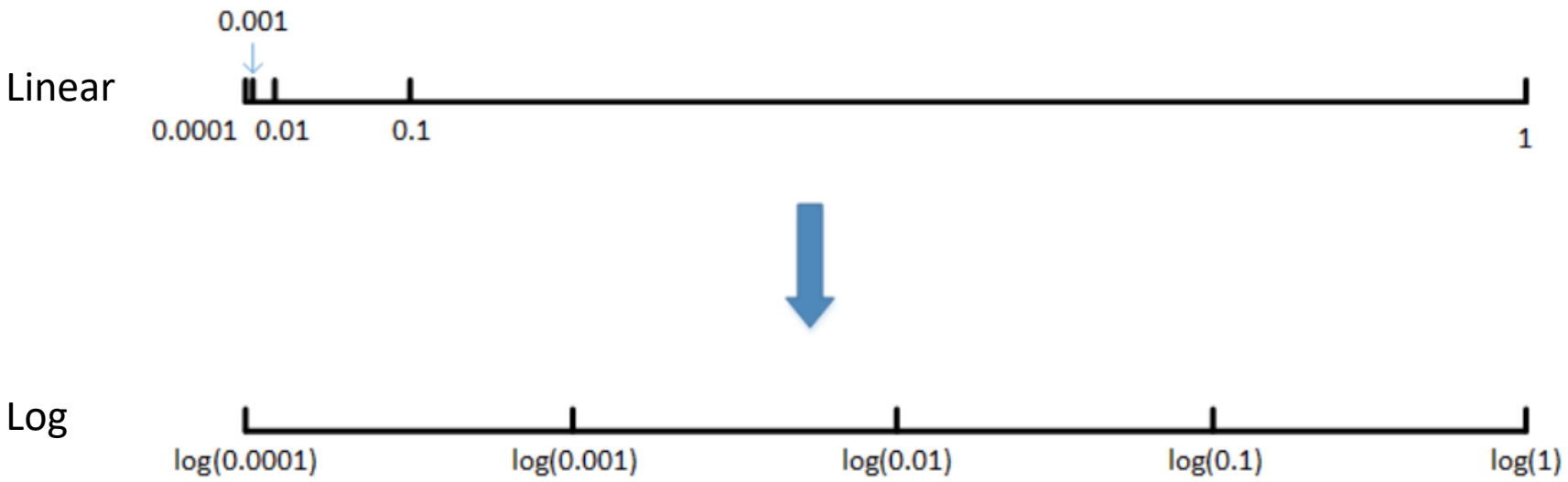


Coarse to fine

Using an Appropriate Scale

#layers, #hidden units → uniform random sample (linear scale)

For other hyperparameter such as α



Linear interval [a,b]

```
m = np.log10(a)
n = np.log10(b)
r = np.random.rand()
r = m + (n-m)*r
r = np.power(10,r)
```

Hyperparameters for exponentially weighted averate

$$\beta = 0.9 \quad \dots \quad 0.999$$

$$\downarrow \qquad\qquad \downarrow$$

$$10 \qquad\quad 1000$$

$$1 - \beta = 0.1 \quad \dots \quad 0.001$$

$$r \in [-3, -1]$$

$$1 - \beta = 10^r$$

$$\beta = 1 - 10^r$$

β: $0.9000 \rightarrow 0.9005$ (0.0005 difference)

$\downarrow$ $\downarrow$

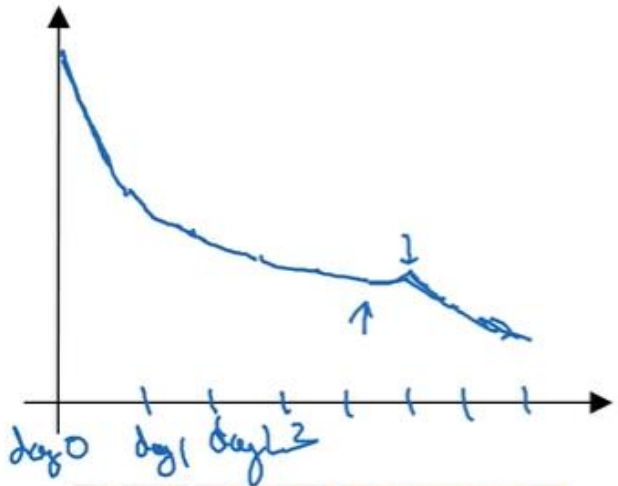
10 10.05

β: $0.9990 \rightarrow 0.9995$ (0.0005 difference)

$\downarrow$ $\downarrow$

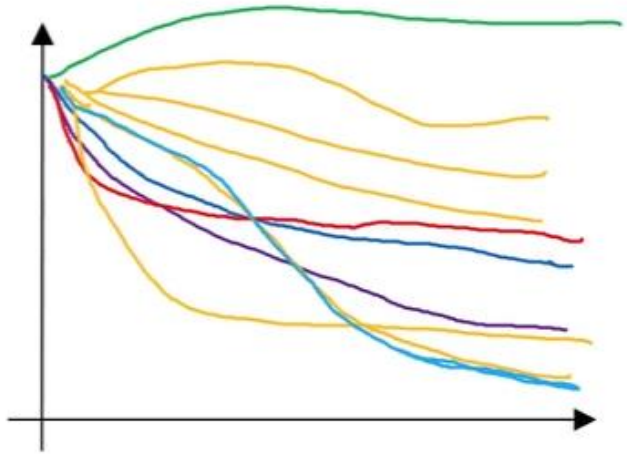
1000 2000

Babysitting one model



Panda

Training many models in parallel



Caviar

Batch Normalization

Batch normalization: Accelerating deep network training by reducing internal covariate shift

S Ioffe, C Szegedy - arXiv preprint arXiv:1502.03167, 2015 - arxiv.org

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and ...

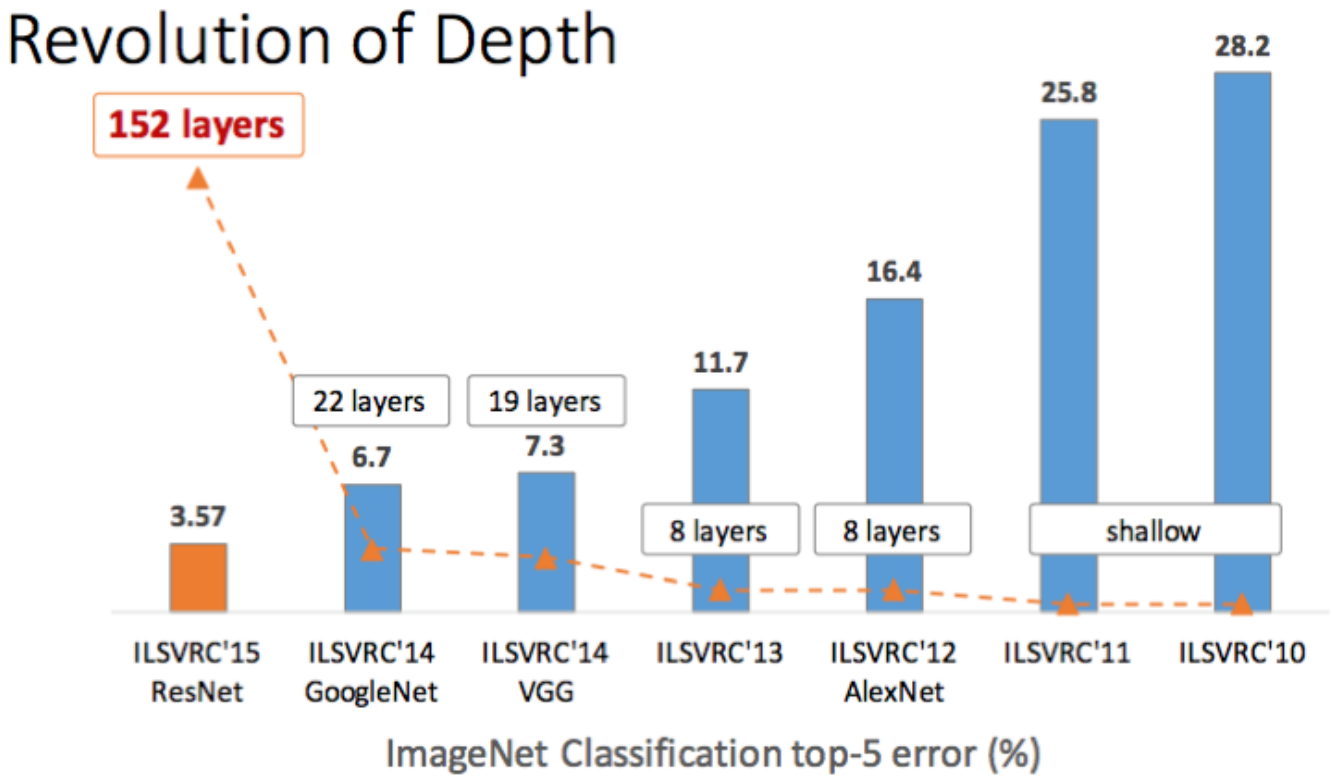
☆ 99 Cited by 19927 Related articles »

Batch normalization

Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

By Jason Brownlee

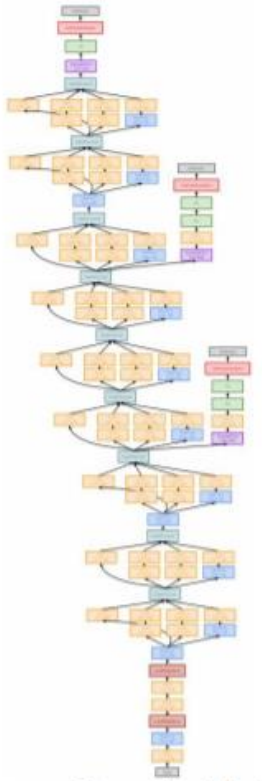
Deep learning before Batch norm



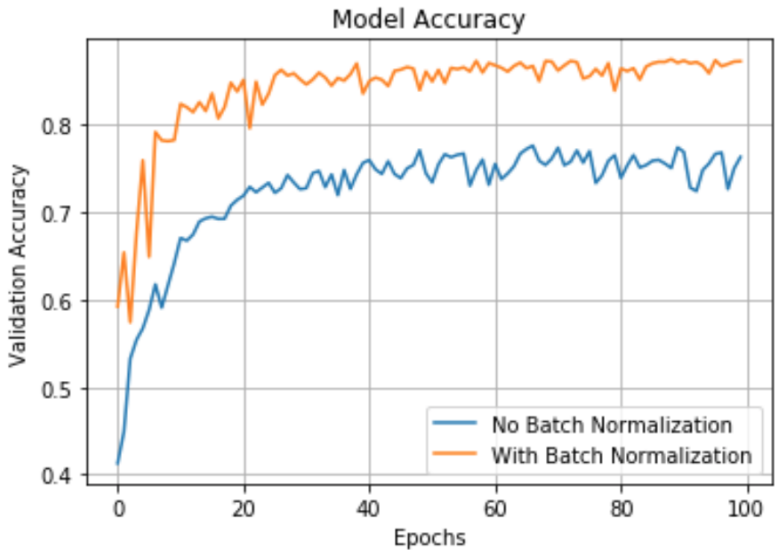
19 layers

VGG16 VGG19

22 layers



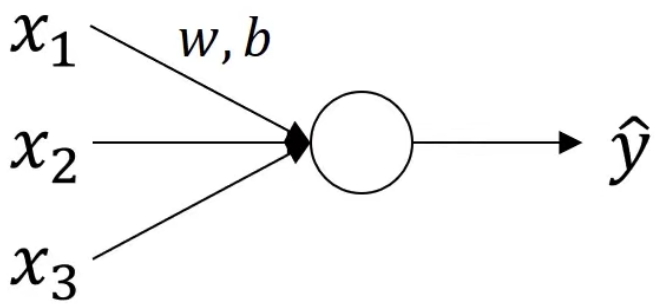
GoogLeNet


Model Accuracy
Validation Accuracy
Epochs
0.4
0.5
0.6
0.7
0.8
0
20
40
60
80
100
No Batch Normalization
With Batch Normalization

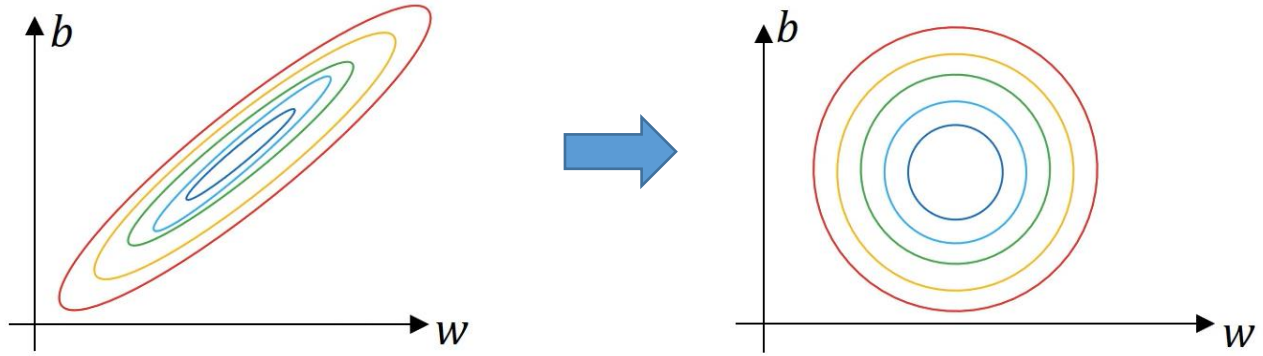

Learning Rate=0.1
Training Accuracy (%)
Steps
50
100
0
5k
10k
15k
Standard
Standard + BatchNorm

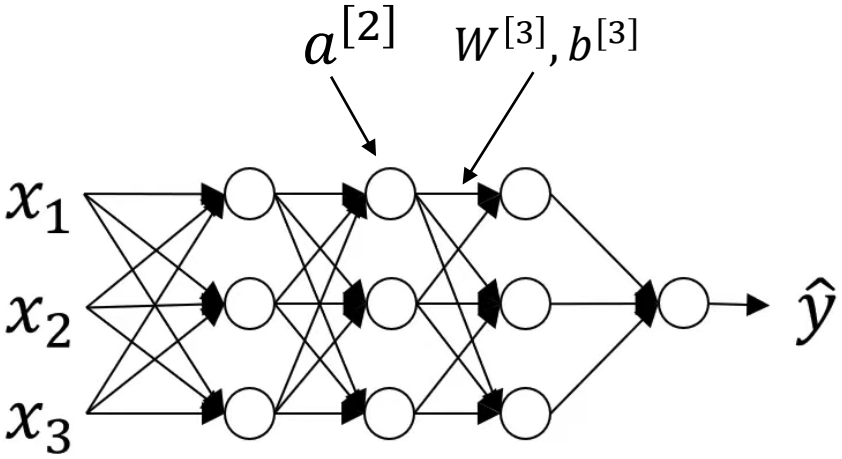

Learning Rate=0.5
Training Accuracy (%)
Steps
50
100
0
5k
10k
15k
Standard
Standard + BatchNorm

Normalizing inputs to speed up learning



$$\mu = \frac{1}{m}\sum_{i=1}^{m} X^{(i)} \qquad \sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(X^{(i)})^2 \qquad X := \frac{X - \mu}{\sigma^2}$$





Can we normalize $a^{[2]}$ to train $W^{[3]}, b^{[3]}$ faster?

→ Normalize $Z^{[2]}$

Implement Batch Normalization

Batch normalize $Z^{[l]} = [z^{[l](1)}, z^{[l](2)}, \dots, z^{[l](m)}]$

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z_i - \mu)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

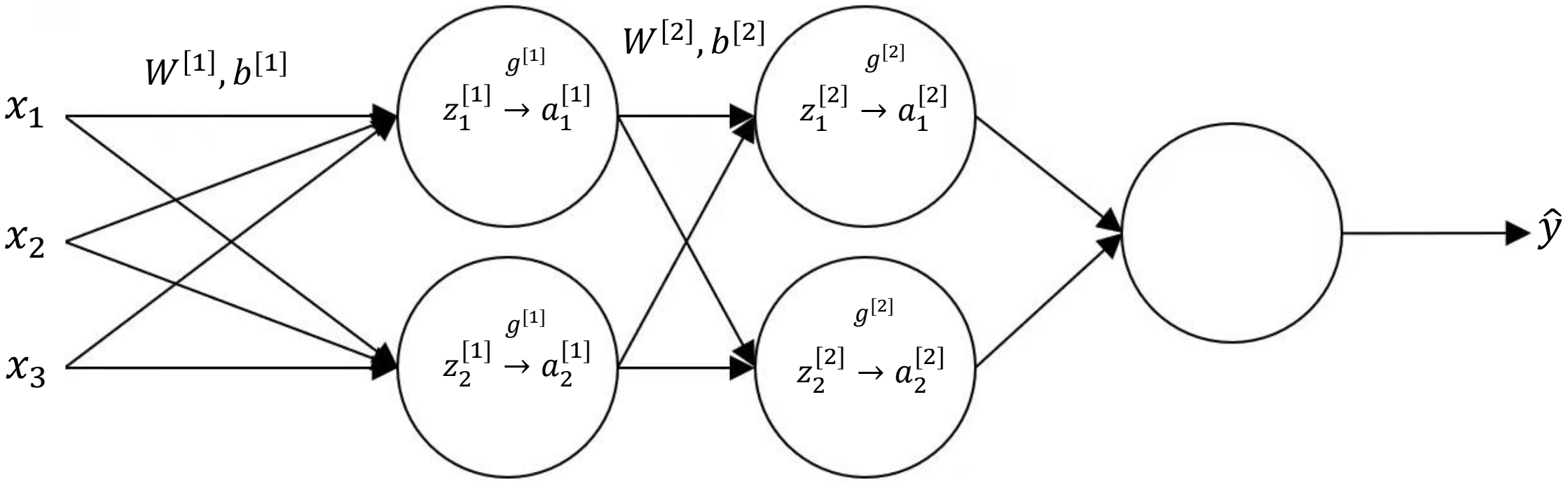
$$\tilde{z}^{(i)} = \gamma \cdot z^{(i)}_{norm} + \beta$$

Learnable parameters

γ and β are used to set whatever mean and variance to $Z^{[l]}$

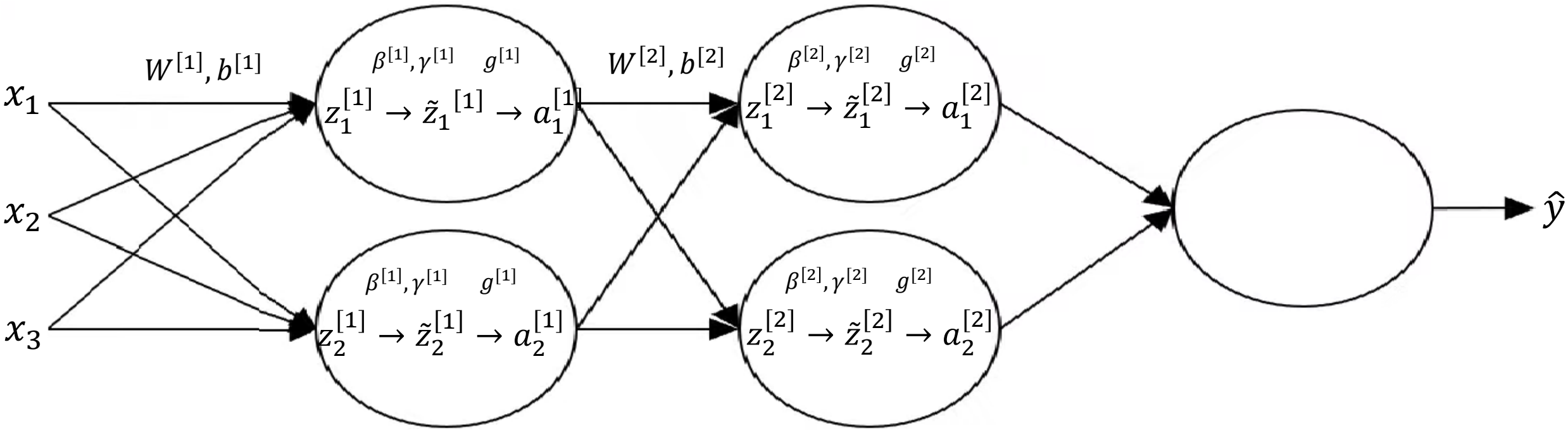
$$\gamma = \sqrt{\sigma^2 + \varepsilon}, \quad \beta = u \quad \Rightarrow \tilde{z}^{(i)} = z^{(i)}$$

Recap normal network



$$X \xrightarrow{W^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow{g^{[1]}} A^{[1]} \xrightarrow{W^{[2]}, b^{[2]}} Z^{[2]} \xrightarrow{g^{[2]}} A^{[2]} \quad \dots$$

Adding batch norm to a network



$$X \xrightarrow{W^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \xrightarrow{g^{[1]}} A^{[1]} \xrightarrow{W^{[2]}, b^{[2]}} Z^{[2]} \xrightarrow{\beta^{[2]}, \gamma^{[2]}} \tilde{Z}^{[2]} \xrightarrow{g^{[2]}} A^{[2]} \dots$$

Parameter: $W^{[1]}, \cancel{b^{[1]}}, W^{[2]}, \cancel{b^{[2]}}, \dots$

$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots$

Update: $W^{[1]} := W^{[1]} - \alpha d_W^{[1]}, \cancel{b^{[1]} := b^{[1]} - \alpha d_b^{[1]}}, \dots$

$\beta^{[1]} := \beta^{[1]} - \alpha d_\beta^{[1]}, \gamma^{[1]} := \gamma^{[1]} - \alpha d_\gamma^{[1]}, \dots$

Implement batch norm with mini-batch

for $t = 1 \ldots num$ (where num is the number of Mini Batch):

- In each $X^{\{t\}}$ perform forward propagation:
  - Use Batch Norm to transform $Z^{\{t\}[l]}$ to $\tilde{Z}^{\{t\}[l]}$
- Use Back propagation to calculate the gradient: $\mathrm{d}_W^{\{t\}[l]}, \mathrm{d}_\gamma^{\{t\}[l]}, \mathrm{d}_\beta^{\{t\}[l]}$
- Update parameters:
  - $w^{[l]} := w^{[l]} - \alpha dw^{[l]}$
  - $\gamma^{[l]} := \gamma^{[l]} - \alpha d\gamma^{[l]}$
  - $\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$

$$Z^{[l]} = W^{[l]} a^{[l-1]} + \cancel{b^{[l]}}$$

$$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \qquad \tilde{z}^{(i)} = \gamma \cdot z_{norm}^{(i)} + \beta$$

Calculate gradient with batch norm

$$X \xrightarrow{W^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \xrightarrow{g^{[1]}} A^{[1]} \xrightarrow{W^{[2]}, b^{[2]}} Z^{[2]} \xrightarrow{\beta^{[2]}, \gamma^{[2]}} \tilde{Z}^{[2]} \xrightarrow{g^{[2]}} A^{[2]} \quad \dots$$

$$da^{[L]} = \frac{\partial L}{\partial a^{[L]}}$$

$$d\tilde{z}^{[L]} = \frac{\partial L}{\partial \tilde{z}^{[L]}} = \frac{\partial L}{\partial a^{[L]}} \frac{\partial a^{[L]}}{\partial \tilde{z}^{[L]}} = da^{[L]} g^{[L]'}(\tilde{z}^{[L]})$$

$$d\beta^{[L]} = \frac{\partial L}{\partial \beta^{[L]}} \; ; \; d\gamma^{[L]} = \frac{\partial L}{\partial \gamma^{[L]}}$$

$$dz^{[L]} = \frac{\partial L}{\partial z^{[L]}}$$

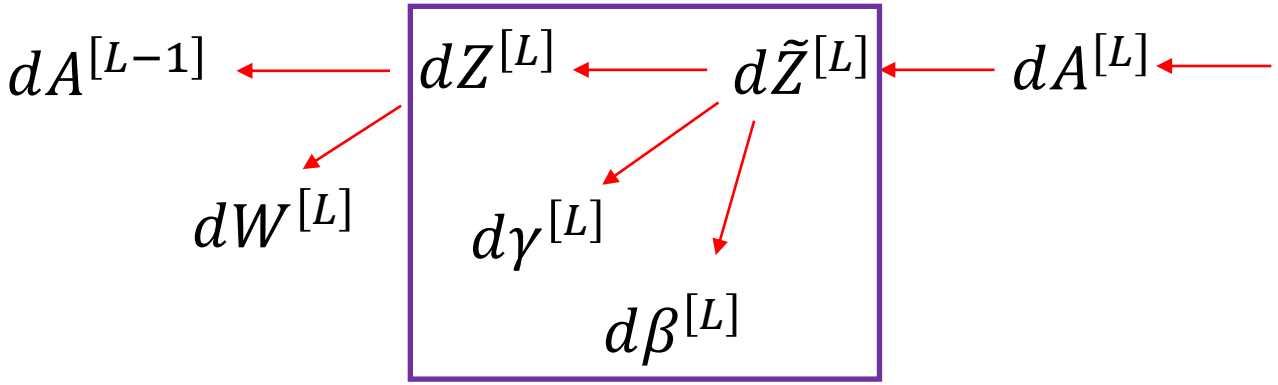
$$dW^{[L]} = \frac{\partial L}{\partial W^{[L]}} = \frac{\partial L}{\partial z^{[L]}} \frac{\partial z^{[L]}}{\partial W^{[L]}} = dz^{[L]} a^{[L-1]^T}$$

$$da^{[L-1]} = \frac{\partial L}{\partial a^{[L-1]}} = \frac{\partial L}{\partial z^{[L]}} \frac{\partial z^{[L]}}{\partial a^{[L-1]}} = W^{[L]^T} dz^{[L]}$$

$$dz^{[L-1]} = \frac{\partial L}{\partial z^{[L-1]}} = \frac{\partial L}{\partial a^{[L-1]}} \frac{\partial a^{[L-1]}}{\partial z^{[L-1]}} = da^{[L-1]} g^{[L-1]'}(z^{[1]}) = W^{[2]^T} dz^{[2]} * g^{[L-1]'}(z^{[L-1]})$$

...

$$\dots \; A^{[L-1]} \xrightarrow{W^{[L]}, b^{[L]}} Z^{[L]} \xrightarrow{\beta^{[L]}, \gamma^{[L]}} \tilde{Z}^{[L]} \xrightarrow{g^{[L]}} A^{[L]} \longrightarrow \mathcal{L}$$



Batch norm derivation

$$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \qquad \tilde{z}^{(i)} = \gamma \cdot z_{norm}^{(i)} + \beta$$

$$dz_{norm}^{(i)[L]} = \frac{\partial L}{\partial z_{norm}^{(i)[L]}} = \frac{\partial L}{\partial \tilde{z}^{[L]}} \frac{\partial \tilde{z}^{[L]}}{\partial z_{norm}^{(i)[L]}} = d\tilde{z}^{[L]}\gamma$$

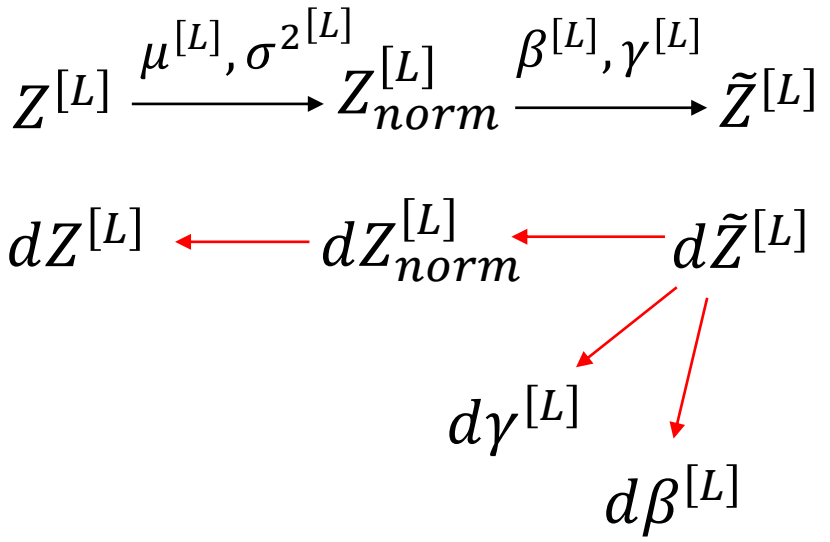
$$d\sigma^{2[L]} = \frac{\partial L}{\partial \sigma^{2[L]}} = \sum_{i=1}^{m} dz_{norm}^{(i)[L]} \left(z^{(i)[L]} - \mu^{[L]}\right)\left(-\frac{1}{2}\right)\left(\sigma^{2[L]} + \varepsilon\right)^{-3/2}$$

$$d\mu^{[L]} = \frac{\partial L}{\partial \mu^{[L]}} = \left(\sum_{i=1}^{m} dz_{norm}^{(i)[L]} \frac{-1}{\sqrt{\sigma^{2[L]} + \varepsilon}}\right) + d\sigma^{2[L]} \frac{\sum_{i=1}^{m} -2\left(z^{(i)[L]} - \mu^{[L]}\right)}{m}$$

$$dz^{(i)[L]} = \frac{\partial L}{\partial z^{(i)[L]}} = dz_{norm}^{(i)[L]} \frac{1}{\sqrt{\sigma^{2[L]} + \varepsilon}} + d\sigma^{2[L]} \frac{2\left(z^{(i)[L]} - \mu^{[L]}\right)}{m} + d\mu^{[L]} \frac{1}{m}$$

$$d\gamma^{[L]} = \frac{\partial L}{\partial \gamma^{[L]}} = \sum_{i=1}^{m} d\tilde{z}^{(i)[L]} z_{norm}^{(i)[L]}$$

$$d\beta^{[L]} = \frac{\partial L}{\partial \beta^{[L]}} = \sum_{i=1}^{m} d\tilde{z}^{(i)[L]}$$



Batch norm at test time

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\tilde{z}^{(i)} = \gamma \cdot z^{(i)}_{norm} + \beta$$

μ, σ^2: estimate using exponentially weighted average (across mini-batch)

$X^{\{1\}}$	$X^{\{2\}}$	$X^{\{3\}}$	...	$X^{\{t\}}$		
$\mu^{\{1\}[l]}$	$\mu^{\{2\}[l]}$	$\mu^{\{3\}[l]}$	...	$\mu^{\{t\}[l]}$	$\rightarrow$	μ
$\mu^{\{1\}[l]}$	$\mu^{\{2\}[l]}$	$\mu^{\{3\}[l]}$	...	$\mu^{\{t\}[l]}$	$\rightarrow$	σ^2

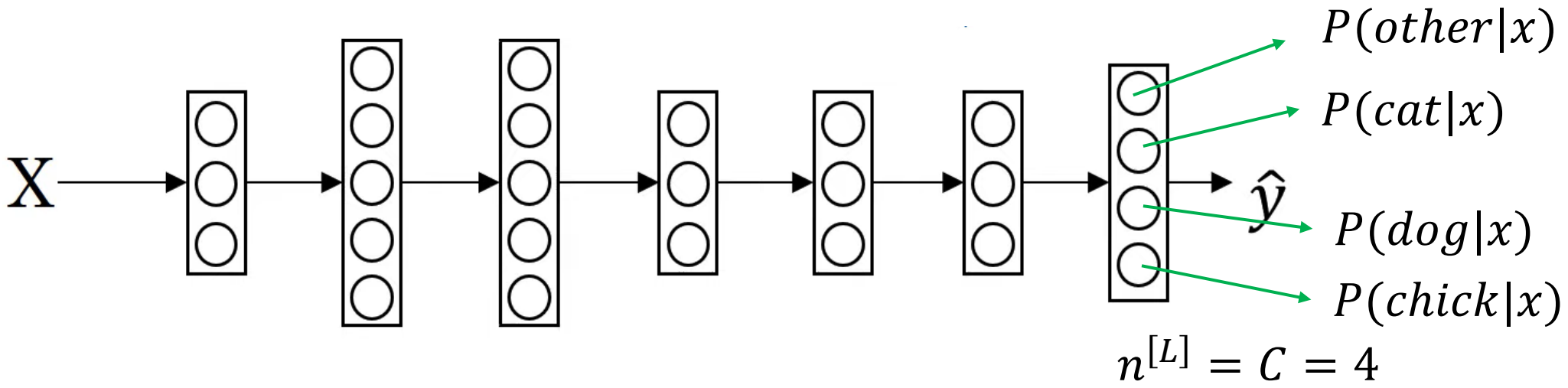
$$z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}} \qquad \tilde{z} = \gamma z_{norm} + \beta$$

Multi-class classification

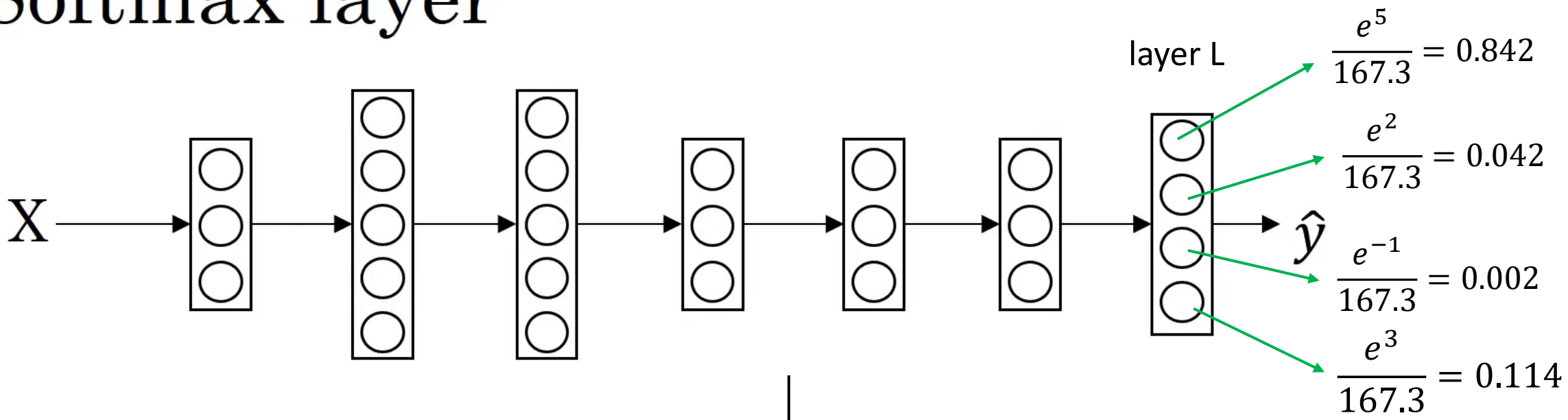
Recognizing cats, dogs, and baby chicks

3 1 2 0 3 2 0 1

$C = \#classes = 4$ (cat, dog, chick, other)



Softmax layer



$$z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$$

Activation function:

$$a_i^{[L]} = \frac{e^{z_i^{[L]}}}{\sum_{i=1}^{C} e^{z_i^{[L]}}} \quad \Rightarrow \quad \sum_{i=1}^{C} a_i^{[L]} = 1$$

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \qquad \sum_{i=1}^{C} e^{z_i^{[L]}} = 176.3$$

$$\hat{y} = a^{[L]} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

Understanding softmax

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \xrightarrow{\text{Hard max}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

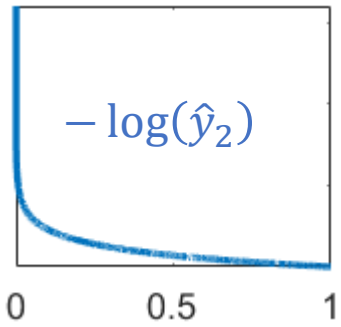
Soft max

$$g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/(e^5 + e^2 + e^{-1} + e^3) \\ e^2/(e^5 + e^2 + e^{-1} + e^3) \\ e^{-1}/(e^5 + e^2 + e^{-1} + e^3) \\ e^3/(e^5 + e^2 + e^{-1} + e^3) \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

Loss function softmax

$$\hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \qquad y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \qquad L(\hat{y}, y) = -\sum_{j=1}^{4} y_j \cdot log\ \hat{y}_j$$

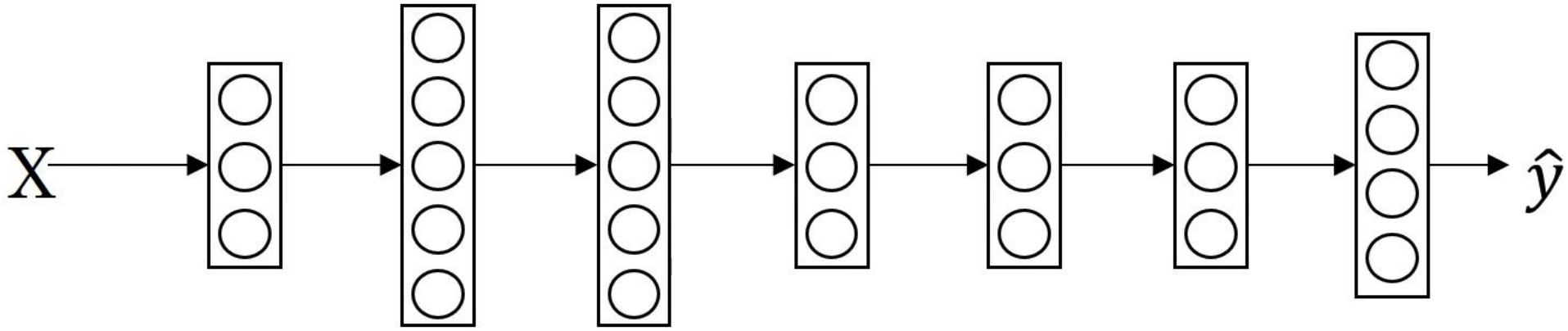
$$L(\hat{y}, y) = -y_2 \cdot log\ \hat{y}_2 = -log\ \hat{y}_2$$



→ $\hat{y}_2$ needs to be close to 1

Cost function on m training example: $J = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y)$

Gradient descent with softmax



$$da^{[L]} = -\frac{1}{a^{[L]}}$$

Backprop:

$$dz^{[L]} = \frac{\partial J}{\partial z^{[L]}} = a_i - y_i$$

$$dZ^{[L]} = A - Y$$

Future topics

- Introduction to Neural Network and Deep Learning
- Improving Deep Neural Networks
- Convolutional Neural Networks
- Sequence Models
- Deep Learning in data mining and big data analysis
- Other architectures and current research activity in deep learning