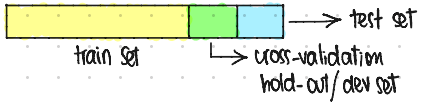
HYPERPARAMETER TUNING, REGULARIZATION AND OPTIMIZATION

COURSE II

PRACTICAL ASPECTS OF DEEP LEARNING

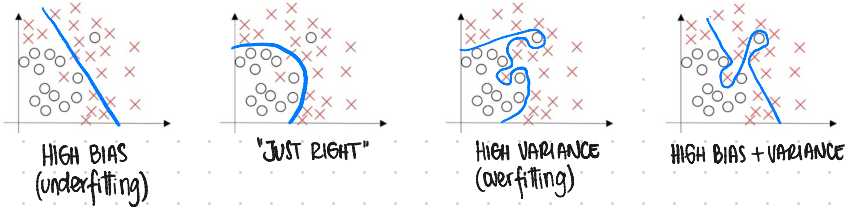
#1. train / developer / test sets:



Before: 70/30% - 60/20/20%

Big data era: 98/1/1% - 99.5/0.4/0.1%

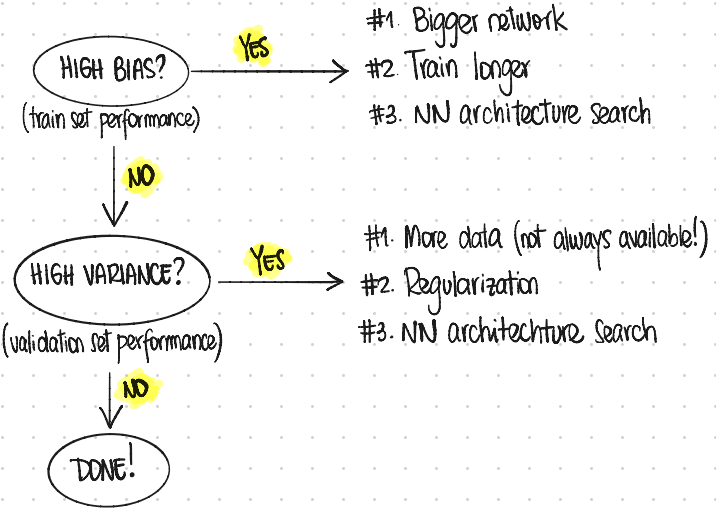
#2. bias vs. variance: is there a trade-off?



train set error:	1%	15%	15%	0.5%
dev set error:	11%	16%	30%	1%
	HIGH VAR.	HIGH BIAS	HIGH BIAS + HIGH VAR.	LOW BIAS + LOW VAR.

WORST NIGHTMARE

BASIC RECIPE FOR MACHINE LEARNING



REGULARIZATION

* prevent/reduce variance (overfitting)
* regularization in logistic regression:

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

REGULARIZATION PARAMETER

L_2 REGULARIZATION

Squared Euclidean norm of w

L_2 REGULARIZATION

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

(most used method)

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

NOT NECESSARY. (b is just one parameter over a large # of parameters)

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

L_1 REGULARIZATION

L_1 REGULARIZATION

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

w will be sparse (with lots of 0) not used very often

✱ regularization in neural networks:

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

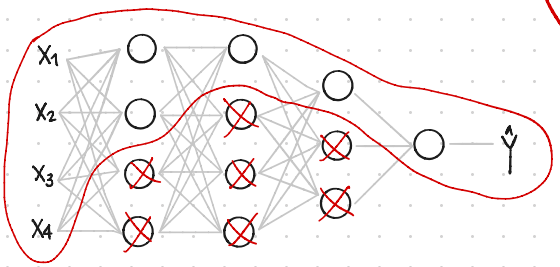
FROBENIUS NORM (L_2 for matrices)

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

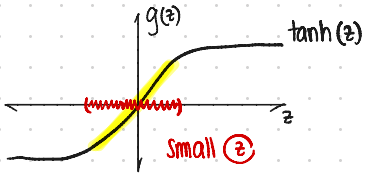
WEIGHT DECAY

✱ why does regularization prevent overfitting?

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



WHEN WE CHOOSE A LARGE λ, WE MAKE $w^{[l]} \approx 0$. THIS REDUCES THE IMPACT OF THE HIDDEN UNITS IN THE NETWORK, "MAKING IT SMALLER." SINCE THE NETWORK IS NOW SMALLER, IT WILL BE ABLE TO GENERALIZE BETTER.

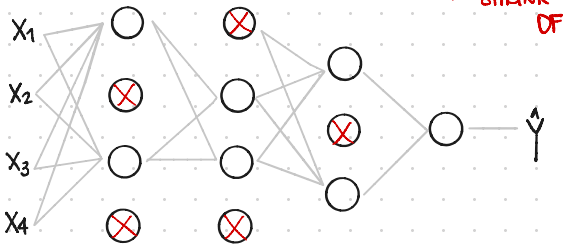


✱ when λ is large, $w^{[l]}$ will be small. Because $z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$ will be small, $g(z)$ will become linear. Thus, every layer will be roughly linear as in linear regression. Even a very deep network will have these "linear" characteristics and will do a much better job preventing overfitting.

DROPOUT REGULARIZATION

* it consists of cancelling some hidden units and make the network "smaller"
* it works because given that the units can't rely on any one feature (it may get knocked out), it has to spread out the weights.

→ SHRINK THE SQUARED NORMS OF THE WEIGHTS



* downside → the cost function $J(w,b)$ is no longer well-defined

OTHER REGULARIZATION METHODS

(#1). data augmentation

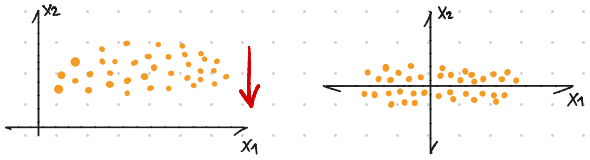
(#2). early stopping → Stop training your network when dev set error > train set error. By stopping half-way, you end up with a mid-size $\|w\|_F^2$.

↘ downside: orthogonalization. By stopping the training you stop the optimalization and also cannot tackle overfitting separetely.

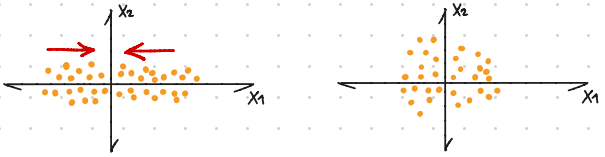
NORMALIZING INPUTS

* speeds up your training
* steps:

(#1). Subtract the mean → $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \Rightarrow x = x - \mu$

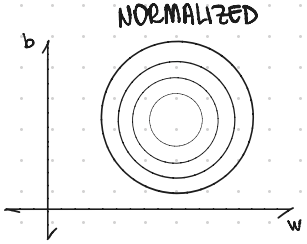


(#2). normalize variance $\rightarrow \sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)2} \Rightarrow x / \sigma^2$



* you need to transform both the train/test set.

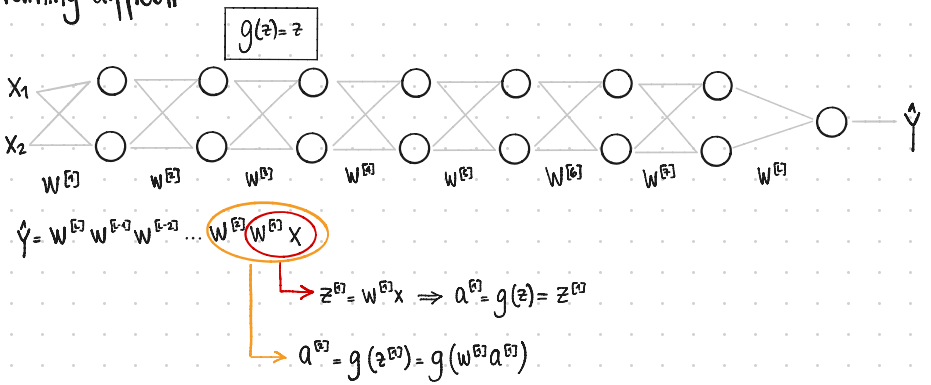
* the gradient descent needs to take more steps to the min.



* the gradient descent takes less steps towards the minimum.

VANISHING/EXPLODING GRADIENTS

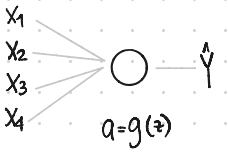
* when you train very deep networks, your slope can get either very big or small, making training difficult.



if $W^{[L]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$, $\hat{y} = W^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} X \approx 1.5^{L-1} \rightarrow \hat{y}$ WILL GROW EXPONENTIALLY AND WILL EXPLODE!!

if $W^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$, $\hat{y} = W^{[l]} \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}^{L-1} X \approx 0.5^{L-1}$ → $\hat{y}$ WILL DECREASE EXPONENTIALLY AND WILL VANISH!!

WEIGHT INITIALIZATION FOR DEEP NETWORKS



$z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b$

LARGER n, SMALLER w_i

$\text{Var}(w_i) = \frac{1}{n} \Rightarrow w^{[l]} = \text{np.random.randn(shape)} * \text{np.sqrt}\left(\frac{2}{n^{[l-1]}}\right)$ ← HYPERPARAMETER

$\text{Var}(w_i) = \frac{2}{n} \Rightarrow$ ReLU activation

$\text{Var}(w_i) = \sqrt{\frac{1}{n^{[l-1]}}} \Rightarrow$ tanh activation (Xavier activation)

GRADIENT CHECKING

* take $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ and reshape it into a big vector θ.

CONCATENATE SO...

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

* take $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$ and reshape it into a big vector $d\theta$.
* for each i:

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

* check: $\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx 10^{-7}$ ✓ GOOD!

$\approx 10^{-3}$ × WORRY!

* implementation notes...

(#1). don't use in training — only to debug

(#2). if algorithm fails the grad check, look at the components to try identify a bug.

(#3). remember regularization!

(#4). doesn't work with dropout (implement grad check without dropout and then do dropout).

(#5). run at random initialization

MINI-BATCH GRADIENT DESCENT

* split the training set into mini-batches

$$X = \underbrace{\left[x^{(1)}\ x^{(2)}\ x^{(3)}\ x^{(4)} \dots x^{(1000)}\right]}_{X^{\{1\}}_{(n_x,\,1000)}} \underbrace{\left|\ x^{(1001)} \dots x^{(2000)}\right|}_{X^{\{2\}}_{(n_x,\,1000)}} \dots x^{(m)}\Big]$$

$$Y = \underbrace{\left[y^{(1)}\ y^{(2)}\ y^{(3)}\ y^{(4)} \dots y^{(1000)}\right]}_{Y^{\{1\}}_{(1,\,1000)}} \underbrace{\left|\ y^{(1001)} \dots y^{(2000)}\right|}_{Y^{\{2\}}_{(1,\,1000)}} \dots y^{(m)}\Big]$$

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

for $t = 1, \dots, 5000$: → # MINI-BATCHES

forward propagation on $X^{\{t\}}$:

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

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

$$\vdots$$

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

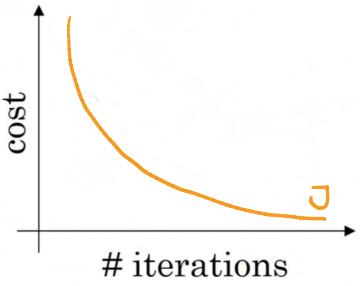
Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$ → EXAMPLES FROM $X^{\{t\}}, Y^{\{t\}}$

→ # EXAMPLES IN THE MINI BATCH

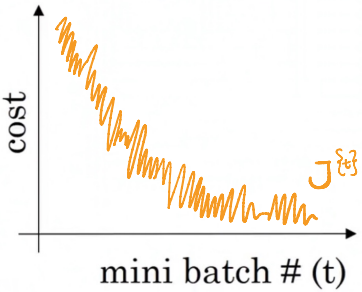
backprop.

* it runs much faster!

BATCH GRADIENT DESCENT



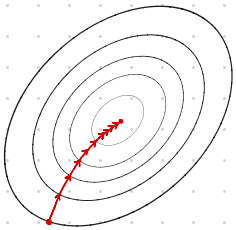
MINI-BATCH GRADIENT DESCENT



if mini-batch size = m (# TRAINING SAMPLES)

BATCH GRADIENT DESCENT

$(X^{\{t\}}, Y^{\{t\}}) = (X, Y)$

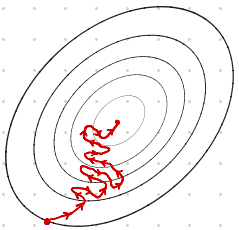


* too long per epoch

if mini-batch size = 1

STOCHASTIC GRADIENT DESCENT

every example is its own mini-batch



* lose all the speedup from vectorization

WHAT WORKS BEST?

↓

SOMETHING IN BETWEEN!

mini-batch size not too big / too small

↓

fastest learning

typical mini-batch size:

64, 128, 256, 512

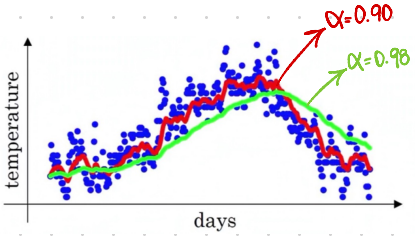
2^6, 2^7, 2^8, 2^9

EXPONENTIALLY WEIGHTED AVERAGES

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

α → CONSTANT

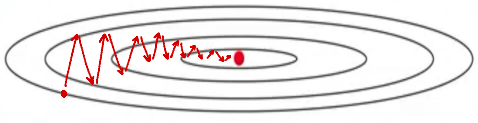
MOVING AVERAGE



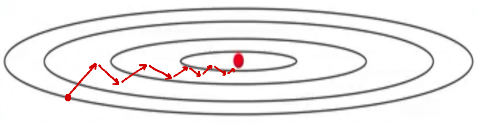
* if α is close to 1, it gives more importance to V_{t-1}
* a smaller α is more sucptible to outliers.

GRADIENT DESCENT WITH MOMENTUM

* compute exponentially weighted averages, and use these averages to compute gradient descent



PLAIN GRADIENT DESCENT



GRADIENT DESCENT + MOMENTUM

* here, you want a slower learning along the vertical axis to prevent overshooting. You also want a faster learning across the horizontal axis. So...

compute dW, db on current mini-batch

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

MOMENTUM TERMS

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

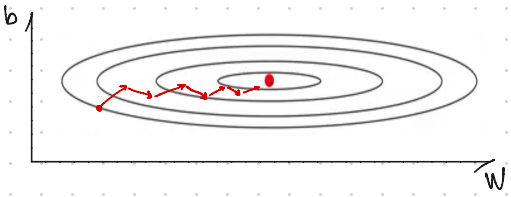
ROOT-MEAN-SQUARED PROP

* compute dw, db on current mini-batch.

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

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

$$W = W - \alpha \frac{dw}{\sqrt{S_{dw}}}, \quad b = b - \alpha \frac{db}{\sqrt{S_{db}}}$$



* these derivatives are much larger in the vertical direction than in the horizontal direction