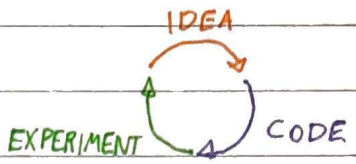
DEEP LEARNING SPECIALIZATION

COURSE 2/5
IMPROVING DEEP NEURAL NETWORKS: HYPERPARAMETER TUNING, REGULARIZATION AND OPTIMIZATION

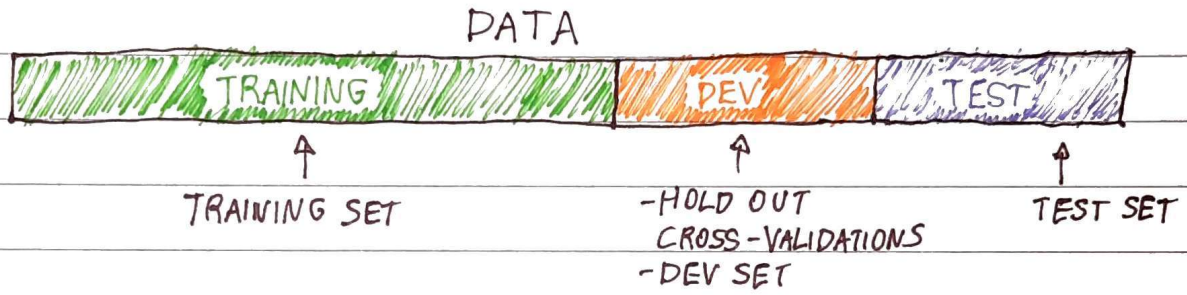
→ TRAIN/DEV/TEST SETS

* APPLIED ML IS A HIGHLY ITERATIVE PROCESS

- TEST HYPERPARAMETERS AND SEE HOW THEY WORK



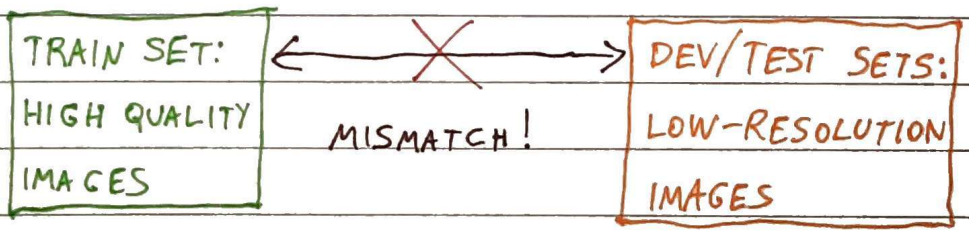
* DATASET ORGANIZATION



- PREVIOUS ERA:
  - → 70% OF DATA FOR TRAINING SET AND 30% FOR TEST
  - → 60% OF DATA FOR TRAINING, 20% FOR DEV, 20% FOR TEST
- BIGDATA ERA:
  - → MUCH SMALLER PERCENTAGES FOR DEV AND TEST

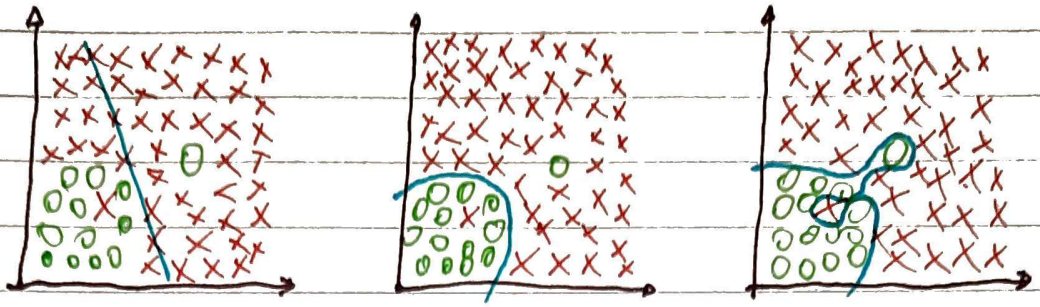
* MISMATCHED TRAIN/TEST DISTRIBUTION

- EXAMPLE



- MAKE SURE DEV AND TEST SETS COME FROM THE SAME DISTRIBUTION.

→ BIAS AND VARIANCE



HIGH BIAS
UNDERFITTING

"JUST RIGHT"

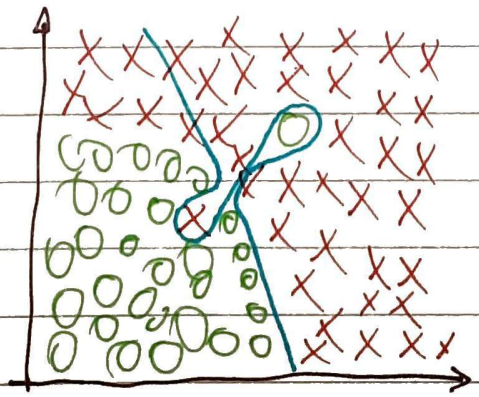
HIGH VARIANCE
OVERFITTING

* HOW TO CHECK BIAS AND VARIANCE ON SYSTEMS WITH MORE THAN TWO VARIABLES? (EXAMPLE)

TRAIN SET ERROR	1%	15%	15%	0,5%
DEV SET ERROR	11%	16%	30%	1%
DIAGNOSIS	HIGH VARIANCE	HIGH BIAS	HIGH BIAS AND HIGH VARIANCE	LOW BIAS AND LOW VARIANCE

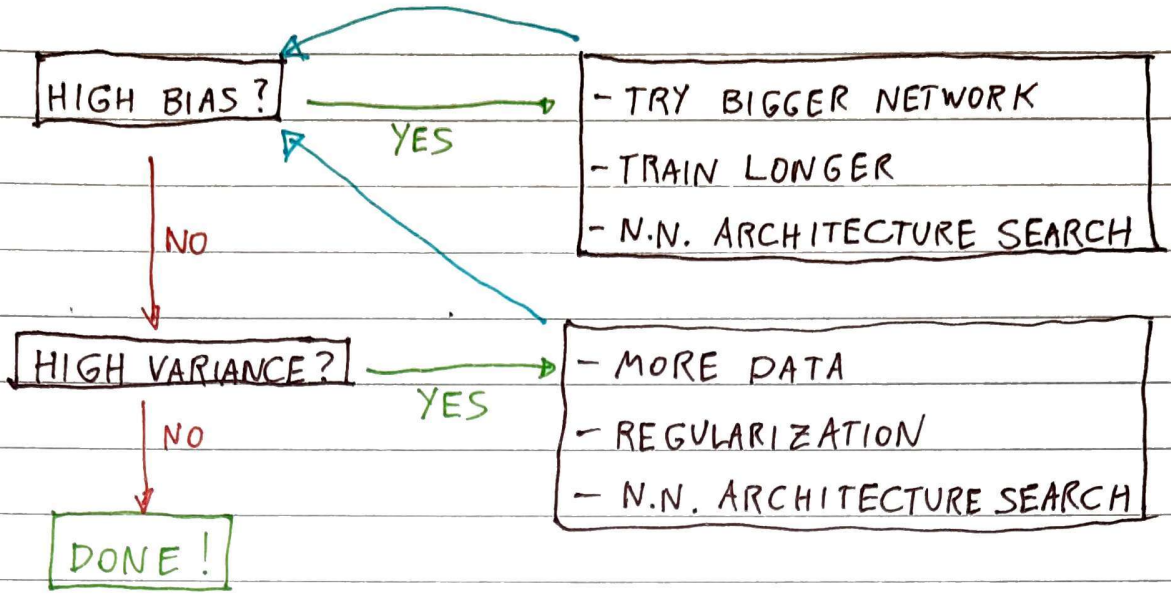
HUMAN ERROR RECOGNIZING A CAT: ≈ 0%
OPTIMAL (BAYES) ERROR: ≈ 0%
} EXAMPLE OF CAT RECOGNIZING

- IN SOME CONDITIONS, BAYES ERROR MAY BE GREATER
- GOAL: GET TRAIN AND DEV SET ERRORS AS CLOSE AS POSSIBLE THAN BAYES ERROR



- HIGH BIAS AND HIGH VARIANCE
- UNDER FIT SOME PART, OVERFIT OTHER PART

→ BASIC RECIPE FOR MACHINE LEARNING



"BIAS - VARIANCE TRADEOFF"

→ REGULARIZATION

* COST FUNCTION.

- $J(w,b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y) + \frac{\lambda}{2m} \|w\|_2^2$
- $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ ← L2 REGULARIZATION
- λ → REGULARIZATION PARAMETER
- $\frac{\lambda}{m} \sum_{i=1}^{m} |w| = \frac{\lambda}{m} \|w\|_1$ ← L_1 REGULARIZATION

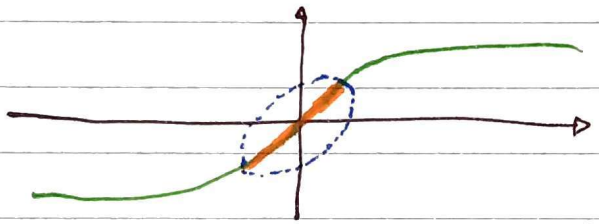
- WITH L1 REGULARIZATION W WILL BE MORE SPARSE, BUT IT DOESN'T HELP MUCH

* REGULARIZATION IN A NEURAL NETWORK

- $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \ldots) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$
- $\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{i,j}^{[l]})^2$ ← FROBENIUS NORM $\|\cdot\|_F^2$

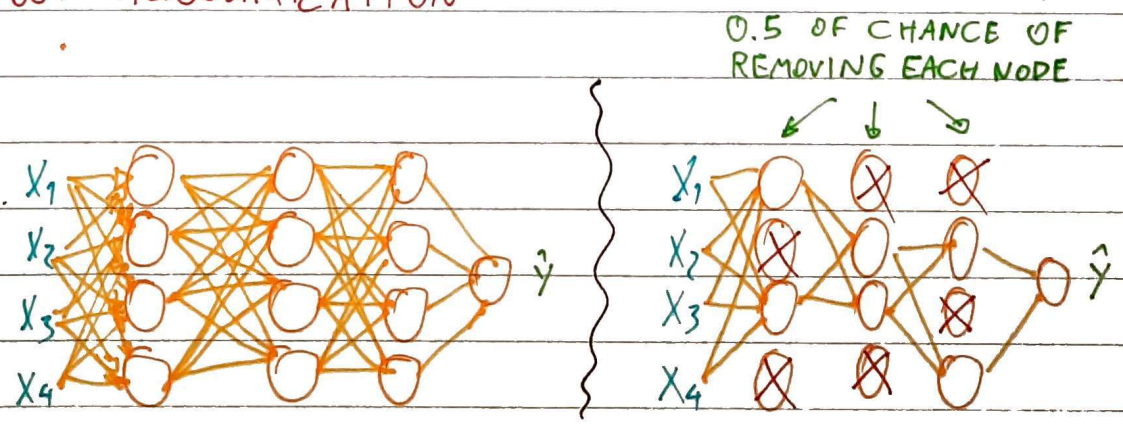
$$dw^{[l]} = (\text{FROM BACK-PROP.}) + \frac{\lambda}{m} \cdot w^{[l]}$$
$$w^{[l]} = w^{[l]} - \alpha \cdot dw^{[l]}$$
$$w^{[l]} = w^{[l]} \cdot \left(1 - \frac{\alpha\lambda}{m}\right) - \alpha \cdot (\text{FROM BACK-PROP.})$$

* HOW DOES REGULARIZATION PREVENT OVERFITTING?



- WITH λ INCREASING, $w^{[l]}$ TENDS TO DECREASE AND THE ACTIVATION FUNCTION GENERALLY WILL BE CALCULATED AT THE REGION THAT TENDS TO BE MORE SIMILAR TO A LINEAR FUNCTION

→ DROPOUT REGULARIZATION



BEFORE DROPOUT

AFTER DROPOUT

→ INVERTED DROPOUT

* ILLUSTRATION WITH LAYER $l = 3$

```
keep_prob = 0.8  # CHANCE OF NODE BE KEPT
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)
a3 /= keep_prob
```

DROPOUT VECTOR →

* MAKING PREDICTIONS AT TEST TIME
  - NO DROPOUT AT TEST TIME
  - THE `/= keep_prob` STATEMENT IS TO NOT IMPLEMENT DROPOUT AT TEST TIME AND DO THE CORRECT SCALING OF ACTIVATIONS, SO THERE IS NO NEED TO RESCALE THE ACTIVATIONS AT TEST TIME

→ WHY DOES DROPOUT WORK?

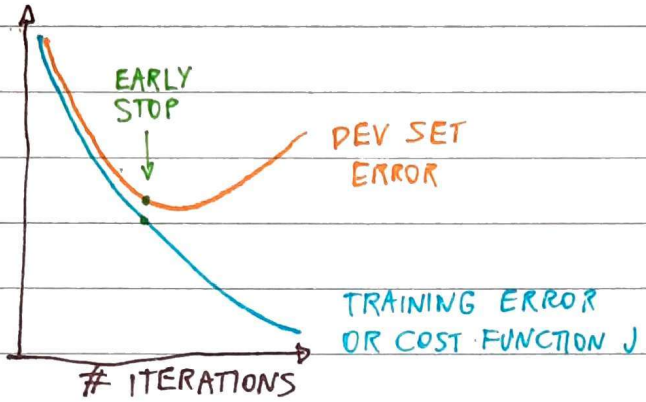
* UNITS CAN'T RELY ON ANY ONE FEATURE, SO HAVE TO SPREAD OUT THE WEIGHTS. ~→ SHRINK WEIGHTS
* WIDE UTILIZATION OF DROPOUT IN COMPUTER VISION
* DOWN SIDE: THE COST-FUNCTION J IS NOT WELL DEFINED ANYMORE, SO YOU DON'T HAVE THIS DEBUG TOOL
  - ALTERNATIVE: TURNOFF DROPOUT, MONITOR COST FUNCTION PLOT, THEN TURN ON DROPOUT

→ OTHER REGULARIZATION METHODS

* DATA AUGMENTATION

- COLLECT MORE DATA (MAY BE TOO EXPENSIVE)
- PROCESS YOUR DATA TO CREATE MORE TRAINING EXAMPLE
  - → EXAMPLE: FOR IMAGES, YOU CAN FLIP IT, ZOOM IT, CROP IT...

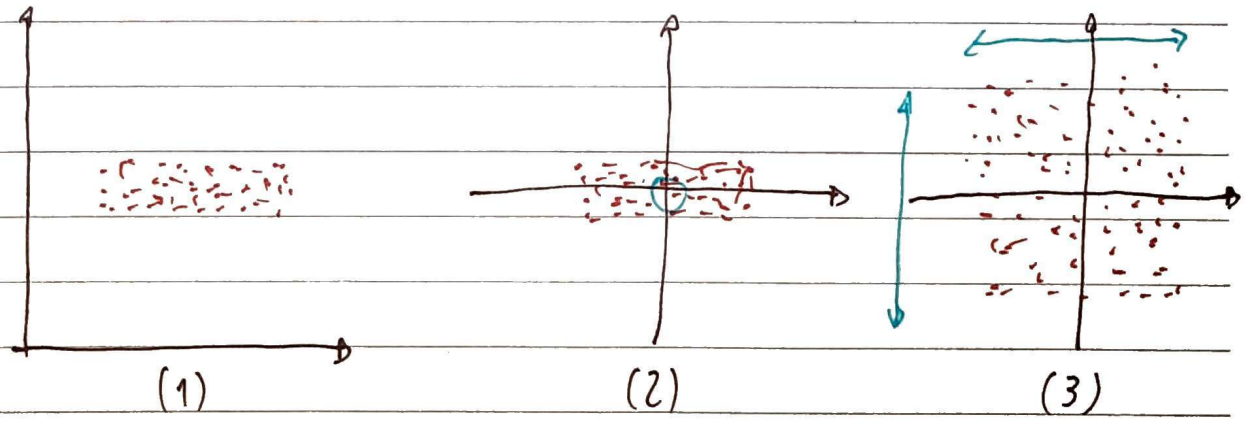
* EARLY STOPPING



- EARLY STOPPING PREVENTS COST FUNCTION AND DEV SET ERRORS TO GET FURTHER AWAY

- DOWN SIDE: IN ORDER TO NOT OVERFIT DATA, IT MAY DEGRADE THE OPTIMIZATION OF THE COST FUNCTION
- IT PREVENTS ORTHOGONALIZATION (CONCEPT THAT WILL BE EXPLAINED LATER)

→ INPUT NORMALIZATION



* EXAMPLE: X WITH 2 FEATURES (SCATTERPLOT)

1 → 2: (SUBTRACT MEAN) $X = X - \mu$ $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$

2 → 3: (NORMALIZE VARIANCE) $X /= \sigma$ $\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)})^2$

ELEMENT-WISE POWER

* WHY USE NORMALIZATION?

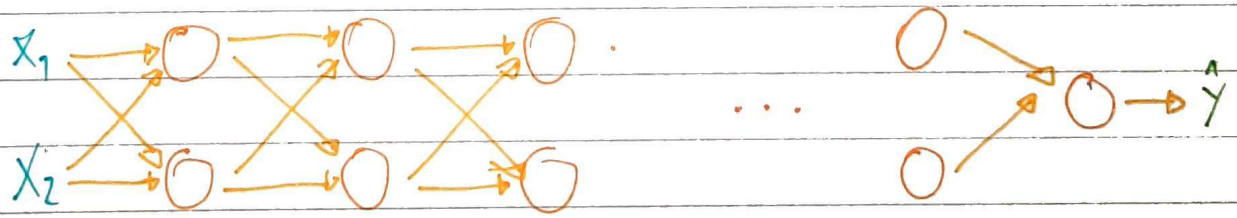
- COST FUNCTION WILL BE OPTIMIZED AND GRADIENT DESCENT WILL TAKE LESS STEPS TO APPROACH THE MINIMUM

* IMPORTANT NOTE

- μ AND σ NEED TO BE THE SAME FOR TRAIN, DEV AND TEST SETS

→ VANISHING / EXPLODING GRADIENTS

* REALLY DEEP N.N. EXAMPLE



- ASSUME $g(z) = z$, $b^{[l]} = 0$, $W^{[l]} = \begin{bmatrix} K & 0 \\ 0 & K \end{bmatrix}$ FOR ALL LAYERS

- $\hat{y} = W^{[L]} W^{[L-1]} W^{[L-2]} \cdots W^{[3]} W^{[2]} W^{[1]} X$

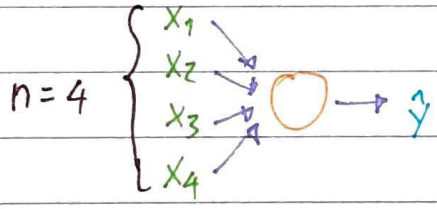
$$\hat{y} = W^{[L]} \cdot \begin{bmatrix} K & 0 \\ 0 & K \end{bmatrix}^{L-1} \cdot X \quad \leftarrow \quad \hat{y} = W^{[L]} \cdot (W^{[L]})^{L-1} \cdot X$$

- IF $K = 0.9$: EXPONENTIAL DECREASE
- IF $K = 1.1$: EXPONENTIAL INCREASE

- WITH VANISHING OR EXPLODING GRADIENTS, IT'LL TAKE A LONG TIME TO GRADIENT DESCENT DECREASE TO MINIMUM COST
- HUGE BARRIER TO TRAIN DEEPER NETWORKS

→ INITIALIZATION OF WEIGHT FOR NEURAL NETWORKS

* SINGLE NEURON EXAMPLE:



FOR THIS EXAMPLE, $b = 0$

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

→ WITH LARGE n: SMALLER w_i

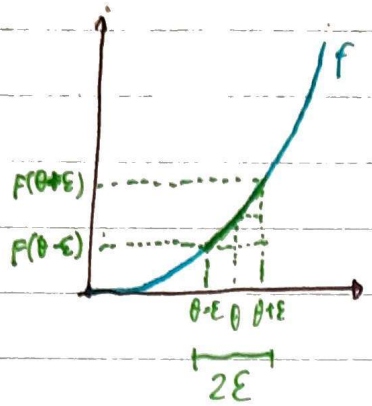
- BETTER CHOICE: $Var(w_i) = \frac{2}{n}$ (RELU), OR $\frac{1}{n}$

- $w^{[l]} = np.random.randn(shape) * np.sqrt(K)$

- RELU: $K = \frac{2}{n^{[l-1]}}$
- TANH: $K = \frac{1}{n^{[l-1]}}$
- BENGIO'S PAPER: $K = \frac{2}{n^{[l-1]} + n^{[l]}}$

→ NUMERICAL APPROXIMATION OF GRADIENTS

* CHECKING DERIVATIVE COMPUTATION



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

→ GRADIENT CHECKING

* TAKE $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ AND CONCATENATE AND RESHAPE TO A BIG VECTOR θ.
* TAKE $dW^{[1]}, dB^{[1]}, \dots, dW^{[L]}, db^{[L]}$, AND CONCATENATE AND RESHAPE TO A BIG VECTOR $d\theta$.
* $J(\theta) = J(\theta_1, \theta_2, \theta_3, \dots \theta_L)$
* CODE

For each i:

$$d\theta_{APPROX}[i] = \frac{J(\theta_1, \theta_2, \dots, \theta_i+\varepsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i-\varepsilon, \dots)}{2\varepsilon}$$

CHECK: $K = \frac{\|d\theta_{APPROX} - d\theta\|_2}{\|d\theta_{APPROX}\|_2 - \|d\theta\|_2} \approx \varepsilon$. IF $K >> \varepsilon$: BUGS.

* IMPLEMENTATION NOTES

- DON'T USE GRAD CHECK IN TRAINING, ONLY ON DEBUG. IT MAY AFFECT PERFORMANCE
- IF ALGORITHM FAILS GRAD CHECK, LOOK AT INDIVIDUAL COMPONENTS IN ORDER TO TRY TO FIND THE CAUSE OF BUG
- IF USING REGULARIZATION, REMEMBER THE TERM ON GRAD CHECK
- GRAD CHECK DOESN'T WORK WITH DROPOUT
- USE RANDOM INITIALIZATION: RUN AGAIN AFTER SOME TRAINING

→ MINIBATCH GRADIENT DESCENT

* COMPUTES GRADIENT DESCENT FASTER ON LARGE DATASETS
* SUBDIVIDE TRAINING SET IN MINOR PARTS

$$X = [\underbrace{X^{(1)}\ X^{(2)}\ X^{(3)} \dots X^{(1000)}}_{X^{\{1\}}}\ \underbrace{X^{(1001)} \dots X^{(2000)}}_{X^{\{2\}}}\ \dots\dots\ \underbrace{X^{(m)}}_{X^{\{5000\}}}]$$

5000 SUBDIVISIONS ON 5000000 DATA

5000 MINIBATCHES OF 1000 EACH

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

$$Y = [\underbrace{y^{(1)}\ y^{(2)} \dots y^{(1000)}}_{y^{\{1\}}}\ \underbrace{y^{(1001)} \dots y^{(2000)}}_{y^{\{2\}}}\ \dots\dots\ \underbrace{y^{(m)}}_{y^{\{5000\}}}]$$

* IMPLEMENTATION

1 EPOCH {

for $t = 1, \dots, 5000$ {

// FORWARD PROP ON $X^{\{t\}}$ } VECTORIZED IMPLEMENTATION (1000 EXAMPLES)

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

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

$\vdots$

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

// COMPUTE COST $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{1000} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_l \|W^{[l]}\|_F^2$

// BACK PROP TO COMPUTE GRADIENT (USING $X^{\{t\}}, y^{\{t\}}$)

$W^{[l]} = W^{[l]} - \alpha \cdot dW^{[l]}$

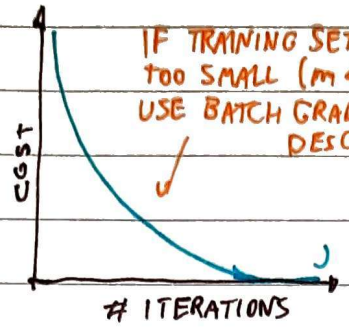
$b^{[l]} = b^{[l]} - \alpha \cdot db^{[l]}$

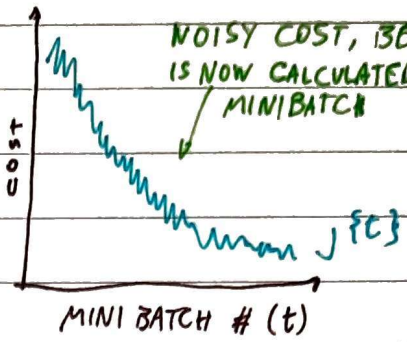
}

- 1 EPOCH: ONE PASS THROUGH THE ENTIRE DATASET CALCULATING GRADIENT DESCENT 5000 TIMES

→ WHY DOES MINIBATCH GRADIENT DESCENT WORK?

* BATCH GRADIENT DESCENT



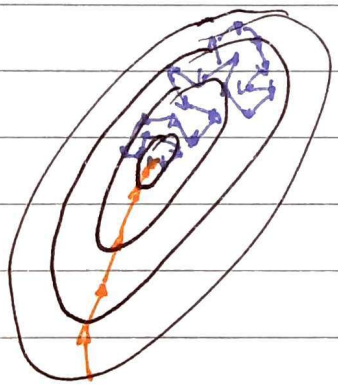


* CHOOSING MINI-BATCH SIZE

IF MINIBATCH SIZE = m: BATCH GRADIENT DESCENT $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

IF MINIBATCH SIZE = 1: STOCHASTIC GRADIENT DESCENT.
EVERY EXAMPLE IS IT'S OWN MINIBATCH. $(X^{\{1\}}, Y^{\{1\}}) = (X^{(1)}, Y^{(1)})$

IN PRACTICE: MINIBATCH SIZE SHOULD BE IN-BETWEEN 1 AND m



COST FUNCTION DIAGRAM REPRESENTATION

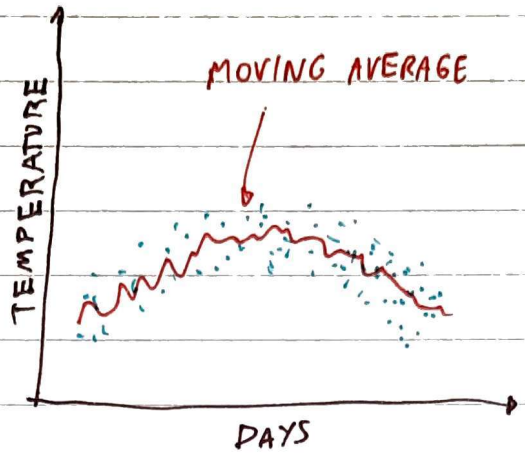
STOCHASTIC	IN-BETWEEN	BATCH
↓	↓	↓
- LOSE SPEEDUP FROM VECTORIZATION	- FASTEST LEARNING - VECTORIZATION - MAKES PROGRESS WITHOUT PROCESS ENTIRE TRAIN SET	- TOO LONG FOR ONE ITERATION

* TYPICAL MINIBATCH-SIZES: 64, 128, 256, 512 (2^n)
* MAKE SURE YOUR MINIBATCH FIT IN YOUR CPU/GPU MEMORY

→ EXPONENTIALLY WEIGHTED MOVING AVERAGES

* EXAMPLE: TEMPERATURE IN LONDON IN ONE YEAR

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

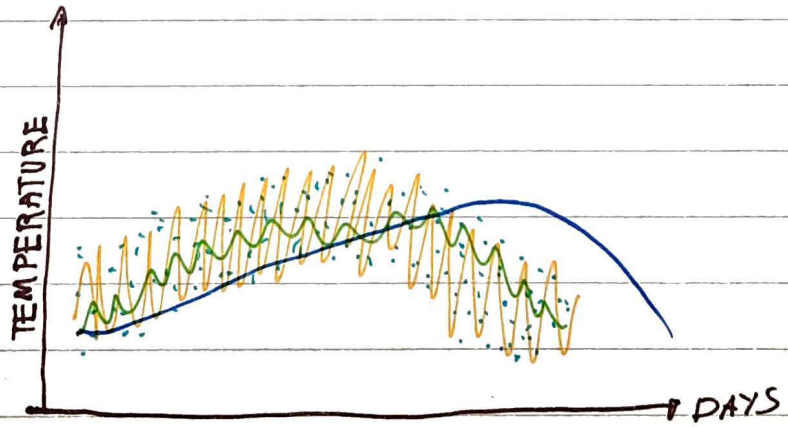


$V_0 = 0$
$V_1 = 0.9 V_0 + 0.1 \theta_1$
$V_2 = 0.9 V_1 + 0.1 \theta_2$
$V_3 = 0.9 V_2 + 0.1 \theta_3 \ldots$
$V_t = 0.9 V_{t-1} + 0.1 \theta_t$

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

- IF $\beta = 0.9$: $\approx$ AVERAGE OF 10 DAYS
- IF $\beta = 0.98$: $\approx$ AVERAGE OF 50 DAYS
- IF $\beta = 0.5$: $\approx$ AVERAGE OF 2 DAYS

- V_t: APPROXIMATED AVERAGE OVER $\approx \frac{1}{1-\beta}$ DAYS



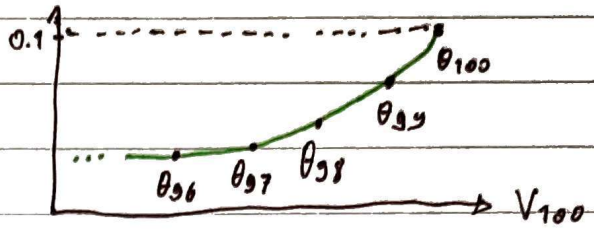
β TOO LARGE: MORE SMOOTH AVERAGE, BUT DELAYED
β TOO SMALL: MINIMUM DELAY, BUT LOT OF NOISE
β "FIT": COMPROMISE BETWEEN SMOOTHNESS AND LOW DELAY

UNDERSTANDING EXPONENTIALLY WEIGHTED AVERAGES

$V_{100} = 0.1\,\theta_{100} + 0.9\,V_{99}$

$V_{99} = 0.1\,\theta_{99} + 0.9\,V_{98}$

$\vdots$



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

$(0.9)^{10} \approx 0.35 \approx \frac{1}{e}$ // $(0.98)^{50} \approx \frac{1}{e}$

$(1-\varepsilon)^{1/\varepsilon} = \frac{1}{e} \longrightarrow \varepsilon = 1-\beta$

IMPLEMENTING EXPONENTIALLY WEIGHTED AVERAGES

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

$\vdots$

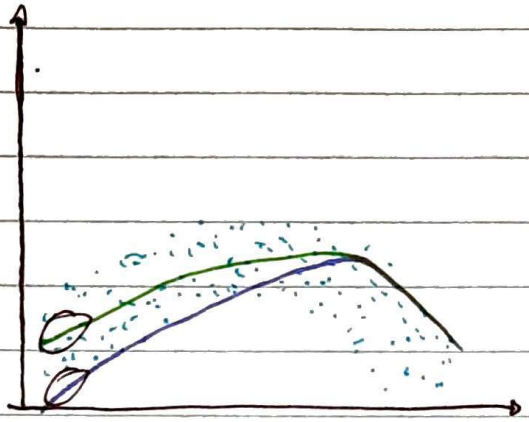
$V_\theta = 0$

REPEAT {

GET NEXT θ_t

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

→ BIAS CORRECTION IN EXPONENTIALLY WEIGHTED AVERAGE



$V_0 = 0$ → AVERAGE IS NOT SO CORRECT AT START (PURPLE LINE)

CORRECTION: USE $V_t = \frac{\beta V_{t-1} + (1-\beta)\theta_t}{1-\beta^t}$

* EXAMPLE

$V_0 = 0$

$V_1 = \cancel{0.98 V_0} + 0.02\theta_1$

$V_2 = 0.98 V_1 + 0.02\theta_2$

$= 0.0196\theta_1 + 0.02\theta_2$

WITH CORRECTION →

$$V_2 = \frac{0.0196\theta_1 + 0.02\theta_2}{1-(0.98)^2}$$

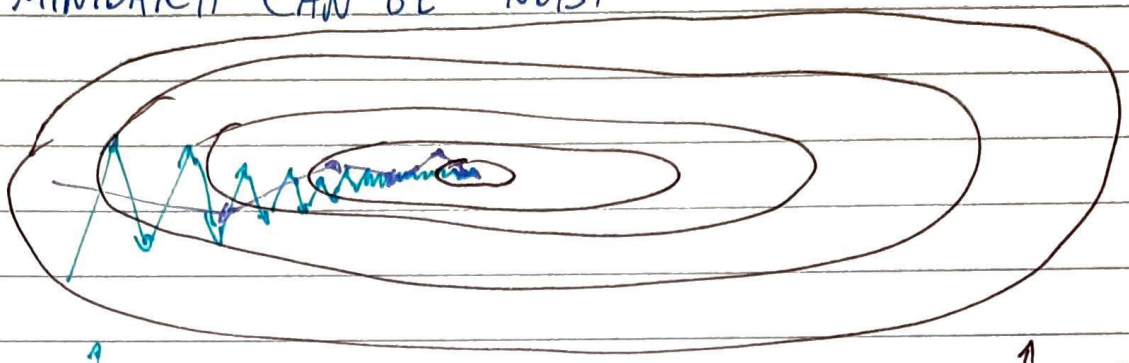
$$V_2 = \frac{0.0196.\theta_1 + 0.02\theta_2}{0.0396}$$

* BIAS CORRECTION PROMOTES A BETTER ESTIMATION AT INITIAL STAGES

* WITH t INCREASING: $\beta^t \to 0$

→ GRADIENT DESCENT WITH MOMENTUM

* MINIBATCH CAN BE "NOISY"



COMMON GRADIENT DESCENT (MINIBATCH)
GRADIENT DESCENT WITH MOMENTUM

↕ SLOWER LEARN
↔ FASTER LEARN

* MOMENTUM CAN IMPROVE α (LEARNING RATE)

* IMPLEMENTATION
  - ON ITERATION t:

    COMPUTE CURRENT dW, db ON CURRENT MINI-BATCH

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

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

    $W = W - \alpha \cdot v_{dW}$

    $b = b - \alpha \cdot v_{db}$

  - BIAS CORRECTION DOESN'T DO MUCH DIFFERENCE, IT'S NOT NEEDED
  - HYPER PARAMETERS: α, β
  - v_{dW} HAS THE SAME DIMENSION OF dW, SO DOES v_{db} WITH db
  - $\beta = 0.9$ IS A GOOD CHOICE
  - v_{dW} AND v_{db} CAN BE IMPLEMENTED WITHOUT $(1-\beta)$ MULTIPLYING dW AND db, LIKE THIS:

    → $v_{dW} = \beta v_{dW} + dW$
    → $v_{db} = \beta v_{db} + db$

    } THIS CAN AFFECT MAGNITUDE OF α

D-2-16

→ ROOT MEAN SQUARE PROPAGATION (RMSPROP)

* ANOTHER TECHNIQUE TO SPEED UP GRADIENT DESCENT
* IMPLEMENTATION
  - ON ITERATION t:

COMPUTE dW, db ON CURRENT MINIBATCH

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

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

$$W = W - \alpha \cdot \frac{dW}{\sqrt{S_{dw}} + \varepsilon}, \qquad b = b - \alpha \cdot \frac{db}{\sqrt{S_{db}} + \varepsilon}$$

dW^2: SMALL, db^2: LARGE, $+\varepsilon$: NUMERICAL STABILITY

→ ADAM OPTIMIZATION (ADAPTIVE MOMENT ESTIMATION)

* RMSPROP + MOMENTUM
* IMPLEMENTATION
  - $V_{dw} = 0$, $S_{dw} = 0$, $V_{db} = 0$, $S_{db} = 0$

ON ITERATION t:

COMPUTE dW, db ON CURRENT MINBATCH

MOMENTUM → $V_{dw} = \beta_1 \cdot V_{dw} + (1-\beta_1)\, dW$, $V_{db} = \beta_1 \cdot V_{db} + (1-\beta_1)\, db$

RMS PROP → $S_{dw} = \beta_2 \cdot S_{dw} + (1-\beta_2)\, dW^2$, $S_{db} = \beta_2 \cdot S_{db} + (1-\beta_2)\, db^2$

BIAS CORRECTION → $V_{dw}^{CORRECTED} = V_{dw} / (1-\beta_1^t)$, $V_{db}^{CORRECTED} = V_{db} / (1-\beta_1^t)$

BIAS CORRECTION → $S_{dw}^{CORRECTED} = S_{dw} / (1-\beta_2^t)$, $S_{db}^{CORRECTED} = S_{db} / (1-\beta_2^t)$

$$W = W - \alpha \cdot \frac{V_{dw}^{CORRECTED}}{\sqrt{S_{dw}^{CORRECTED}} + \varepsilon}, \qquad b = b - \alpha \cdot \frac{V_{db}^{CORRECTED}}{\sqrt{S_{db}^{CORRECTED}} + \varepsilon}$$

* HYPER PARAMETER COMMON CHOICES

α: NEEDS TO BE TUNED

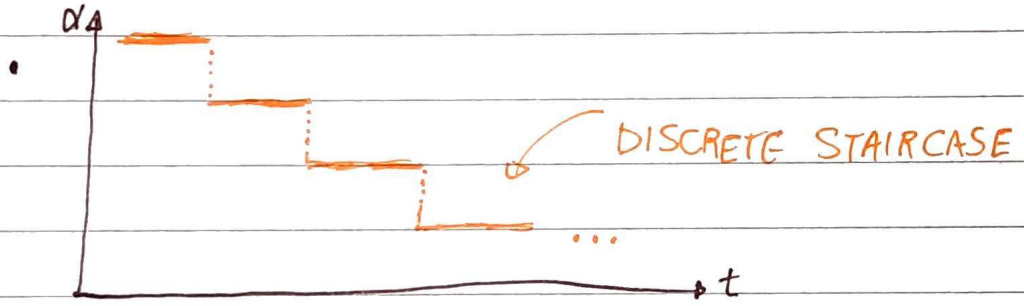
β_1: 0.9

ε: 10^{-8}

β_2: 0.999

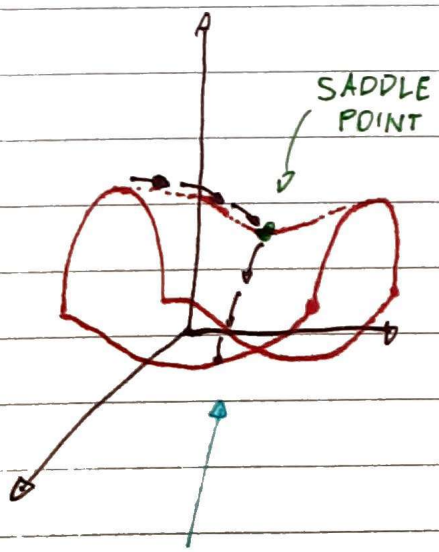
→ LEARNING DECAY

* DECREASE ALPHA AT EVERY EPOCH, TO HELP THE TRAINING CONVERGE TO MINIMUM, DESPITE THE "NOISE" ON MINI-BATCH
* SOME LEARNING DECAY ALTERNATIVES:
  - $\alpha = \frac{\alpha_0}{1 + \text{decay\_rate} \cdot \text{epoch\_num}}$
  - $\alpha = \text{decay\_rate}^{\text{epoch\_num}} \cdot \alpha_0$ ← FOR THIS, decay_rate < 1
  - $\alpha = \frac{K}{\sqrt{\text{epoch\_num}}} \cdot \alpha_0$ OR $\alpha = \frac{K}{\sqrt{t}} \cdot \alpha_0$



- MANUAL DECAY IS ANOTHER OPTION

→ LOCAL OPTIMA AND IT'S PROBLEMS



PRACTICAL CASES END UP IN SOME CURVES "SADDLE-LIKE"

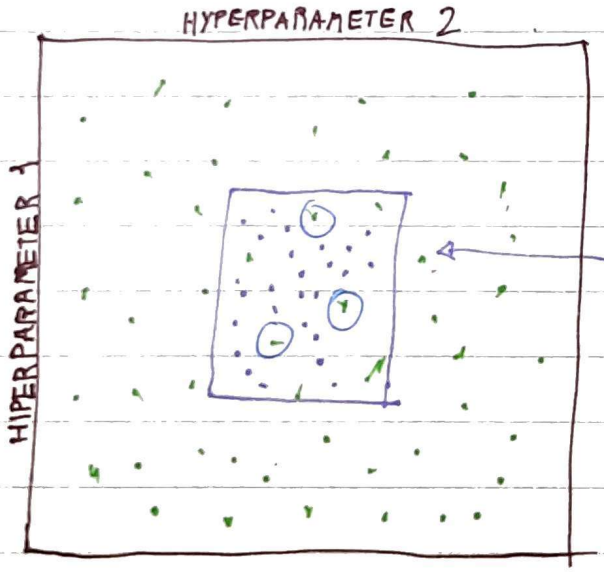
- TO CONVERGE TO A SADDLE POINT, GRADIENT DESCENT CAN BE SLOW
- IN REAL PROBLEMS, IT'S UNLIKELY TO GET STUCK ON LOCAL OPTIMA
- FOR PLATEAUS AND SADDLE POINTS, OPTIMIZATION ALGORITHMS CAN HELP TO SPEED UP CONVERGENCE

→ HYPER PARAMETERS TUNING PROCESS

* ORDER OF IMPORTANCE

  - α → BIGGEST PRIORITY
  - β (MOMENTUM)
    # HIDDEN UNITS ON EACH LAYER
    MINI-BATCH SIZE
    → MEDIUM PRIORITY
  - # LAYERS
    LEARNING RATE DECAY
    → MINOR PRIORITY
  - $\beta_1, \beta_2, \varepsilon$ (ADAM)
    → NOT A PRIORITY AT ALL
    (JUST USE STANDARD VALUES)

* TRY RANDOM VALUES: DON'T USE A GRID
* TUNING COARSE TO FINE (EXAMPLE)



→ APPROPRIATE SCALE FOR HYPERPARAMETER TUNING

* CHANGE THEM ON LOG SCALE WILL MAKE MORE DIFFERENCE THAN ON LINEAR SCALE
* FOR EXPONENTIALLY WEIGHT AVERAGES: LOG SCALE FOR $\beta \to 1$

→ HYPERPARAMETER TUNING IN PRACTICE

* "BABYSITTING" A MODEL (PANDA)
  - GOOD FOR LOW COMPUTATIONAL RESOURCES
  - TRAIN A MODEL AND TWEAK HYPERPARAMETERS TO SE HOW IT PERFORM
* TRAINING MANY MODELS IN PARALLEL (CAVIAR)
  - REQUIRES MORE COMPUTATIONAL RESOURCES
  - TRAIN MANY MODELS SIMULTANEOUSLY WITH DIFFERENT HYPERPARAMETER SETTINGS AND SEE WHICH ONE PERFORMS BETTER

→ BATCH NORMALIZATION

* NORMALIZING INPUT SPEEDS UP THE LEARNING
* WHY NOT NORMALIZE THE LAYER ACTIVATIONS?
* KEY IDEA: NORMALIZE $Z^{[l]}$ TO TRAIN $W^{[l+1]}$, $b^{[l+1]}$ FASTER
* IMPLEMENTATION
  - GIVEN SOME INTERMEDIATE VALUES $Z^{[l](i)}$
    - $\mu = \frac{1}{m} \sum_i z^{(i)}$
    - $\sigma^2 = \frac{1}{m} \sum_i (z_i - \mu)^2$
    - $Z^{(i)}_{NORM} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$
    - $\tilde{Z}^{(i)} = \gamma \cdot Z^{(i)}_{NORM} + \beta$

      LEARNABLE PARAMETERS, LIKE W AND b
    - USE $\tilde{Z}^{[l](i)}$ INSTEAD OF $Z^{[l](i)}$
  - IF $\gamma = \sqrt{\sigma^2 + \varepsilon}$ AND $\beta = \mu$: $\tilde{Z}^{(i)} = Z^{(i)}$

→ ADDING BATCH NORM TO A N.N.

* $X \xrightarrow[w^{[1]}, b^{[1]}]{} z^{[1]} \xrightarrow[\gamma^{[1]}, \beta^{[1]}]{\text{B.N.}} \tilde{z}^{[1]} \rightarrow a^{[1]} \xrightarrow[b^{[2]}]{w^{[2]}} z^{[2]} \xrightarrow[\gamma^{[2]}, \beta^{[2]}]{\text{B.N.}} \tilde{z}^{[2]} \rightarrow a^{[2]} \rightarrow \dots$

* PARAMETERS: $w^{[l]}$, ~~$b^{[l]}$~~, $\gamma^{[l]}$, $\beta^{[l]}$

$b^{[l]}$ WILL BE CANCELLED BY MEAN SUBTRACTION STEP, WE DON'T NEED IT WHEN USING BATCH NORM.

↳ THIS BETA IS NOT THE SAME AS BETA FROM MOMENTUM, RMSPROP OR ADAM OPTIMIZER

* POSSIBILITY TO WORK WITH MINI-BATCHES

* DIMENSIONS:

$\beta^{[l]}: (n^{[l]}, 1)$ $\qquad$ $\gamma^{[l]}: (n^{[l]}, 1)$

* IMPLEMENTATION

FOR $t = 1, \dots,$ numMiniBatches:

COMPUTE FWD-PROP ON $X^{\{t\}}$

INSIDE FWD-PROP. → IN EACH HIDDEN LAYER, USE B.N. TO RETRIEVE $\tilde{z}^{[n]}$

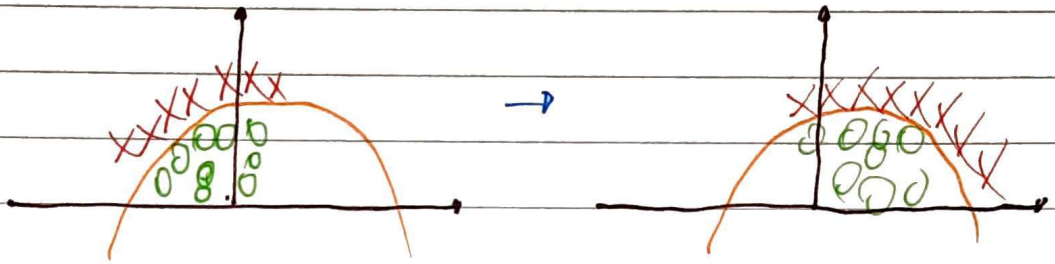
- USE BACK-PROP TO COMPUTE $dw^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$
- UPDATE PARAMETERS:

$$w^{[l]} = w^{[l]} - \alpha \cdot dw^{[l]}$$
$$\beta^{[l]} = \beta^{[l]} - \alpha \cdot d\beta^{[l]}$$
$$\gamma^{[l]} = \gamma^{[l]} - \alpha \cdot d\gamma^{[l]}$$

- BATCH NORMALIZATION WORKS WELL WITH MOMENTUM, RMSPROP AND ADAM.

→ WHY DOES BATCH NORM WORK?

* SHIFTING INPUT DISTRIBUTION (EXAMPLE)
  - NEURAL NETWORK TO RECOGNIZE CATS TRAINED ONLY WITH BLACK CATS MAY NOT WORK PROPERLY ON COLORED CATS
  - "COVARIATE SHIFT"



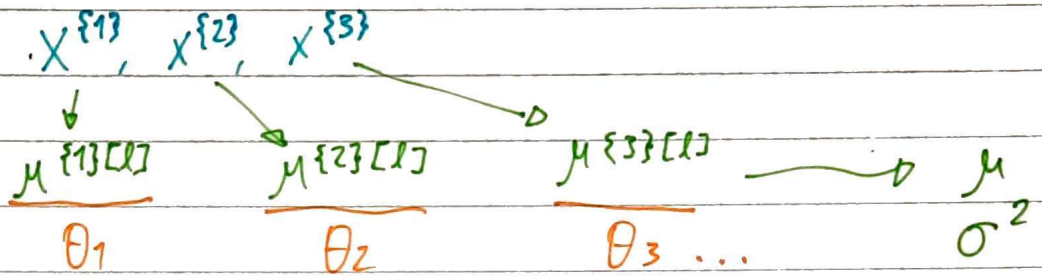
* WHY THIS IS A PROBLEM WITH NEURAL NETWORK?
  - COVARIATE SHIFT SPREADS OVER THE LAYERS
* BATCH NORM KEEPS MEAN = 0 AND VARIANCE = 1 IN EVERY SINGLE LAYER
* INPUT DISTRIBUTION DOESN'T CHANGE TOO MUCH WITH BATCH NORMALIZATION
* SPEEDS UP TRAINING
* BATCH NORM ALSO WORKS AS REGULARIZATION

→ BATCH NORM AS REGULARIZATION

* EACH MINI-BATCH IS SCALED BY THE MEAN/VARIANCE COMPUTED ON JUST THAT MINIBATCH.
* THIS ADDS NOISE TO $z^{[l]}$ WITHIN THAT MINI-BATCH. SIMILAR TO DROPOUT, IT ADDS NOISE TO EACH HIDDEN LAYERS' ACTIVATIONS
* THIS HAS A SLIGHT REGULARIZATION EFFECT
* THE LARGER THE MINIBATCH, THE SMALLER THE NOISE, AND THE SMALLER THE REGULARIZATION EFFECT

→ BATCH NORM AT TEST TIME

* ESTIMATE μ, σ^2 USING EXPONENTIALLY WEIGHTED AVERAGE ACROSS MINI-BATCHES



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

→ DERIVATIVES FOR BATCH NORM (IOFFE & SZEGEDY, 2015)

$$\bullet \frac{\partial L}{\partial z^{(i)}_{norm}} = \frac{\partial L}{\partial \tilde{z}^{(i)}} \cdot \gamma$$

$$\bullet \frac{\partial L}{\partial \sigma^2} = \sum_{i=1}^{m} \frac{\partial L}{\partial z^{(i)}_{norm}} \cdot (z^{(i)} - \mu) \cdot \left(\frac{-(\sigma^2 + \varepsilon)^{-\frac{3}{2}}}{2} \right)$$

$$\bullet \frac{\partial L}{\partial \mu} = \left(\sum_{i=1}^{m} \frac{\partial L}{\partial z^{(i)}_{norm}} \cdot \frac{-1}{\sqrt{\sigma^2 + \varepsilon}} \right) + \frac{\partial L}{\partial \sigma^2} \cdot \left(\frac{-2}{m} \right) \cdot \sum_{i=1}^{m} (z^{(i)} - \mu)$$

$$\bullet \frac{\partial L}{\partial z^{(i)}} = \frac{\partial L}{\partial z^{(i)}_{norm}} \cdot \frac{1}{\sqrt{\sigma^2 + \varepsilon}} + \frac{\partial L}{\partial \sigma^2} \cdot \frac{2(z^{(i)} - \mu)}{m} + \frac{\partial L}{\partial \mu} \cdot \frac{1}{m}$$

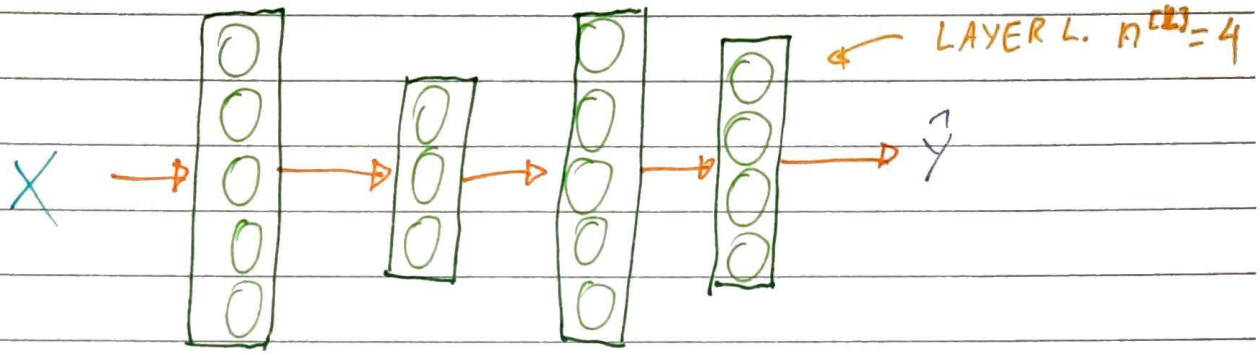
$$\bullet \frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial \tilde{z}^{(i)}} \cdot z^{(i)}_{norm}$$

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

→ SOFTMAX REGRESSION

* CLASSIFICATION WITH MULTIPLE CLASSES (MORE THAN 2)

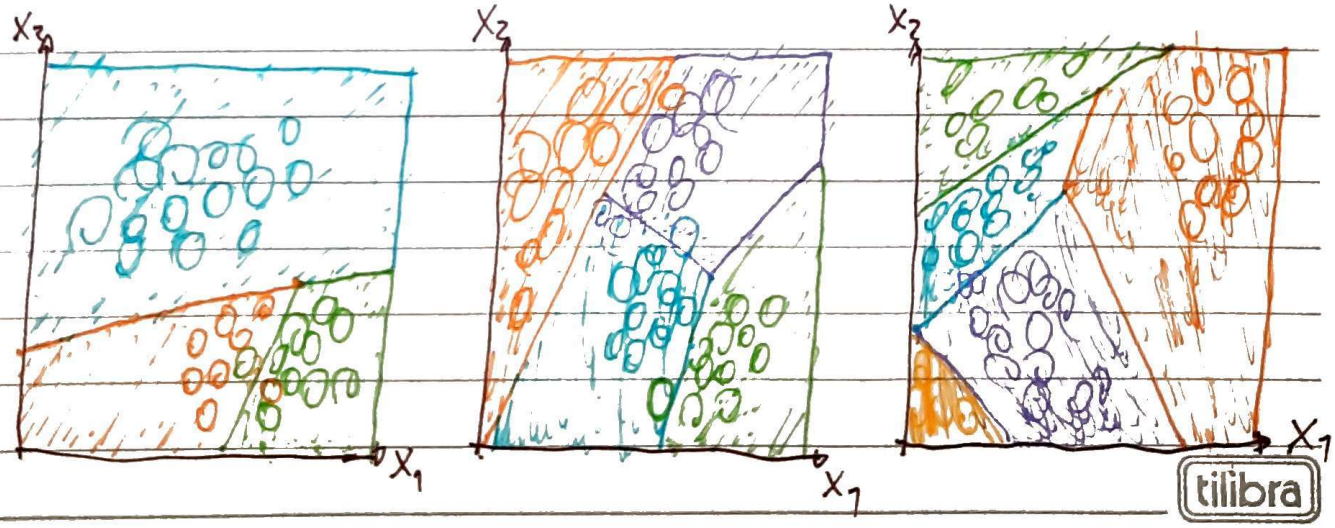
- ASSUMING C THE NUMBER OF POSSIBLE CLASSES:
  - → $n^{[l]} = C$
- EACH OUTPUT REPRESENTS PROBABILITY OF A CLASS
- THE SUM OF ALL OUTPUTS IS 1 (100%)



* OUTPUT: SOFTMAX LAYER

- $z^{[L]} = w^{[L]} \cdot a^{[L-1]} + b^{[L]}$
- ACTIVATION FUNCTION
  - → $t = e^{(z^{[L]})}$
  - → $a^{[L]} = \frac{e^{(z^{[L]})}}{\sum_{i=1}^{C} t_i}$ → $a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{C} t_j}$

* SOFTMAX EXAMPLES



→ TRAINING A SOFTMAX LAYER

* SOFTMAX REGRESSION GENERALIZES LOGISTIC REGRESSION TO C CLASSES
* IF C=2, SOFTMAX REDUCES TO LOGISTIC REGRESSION
* LOSS FUNCTION
  - $L(\hat{y}, y) = -\sum_{j=1}^{n^{[L]}} y_j \cdot \log(\hat{y}_j)$
* COST FUNCTION
  - $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \ldots) = \frac{1}{m} \cdot \sum_{i=1}^{n^{[L]}} L(\hat{y}^{(i)}, y^{(i)})$
* DIMENSIONS OF OUTPUT AND PREDICTIONS:

  $Y.shape = (C, m)$ $\quad\quad$ $\hat{Y}.shape = (C, m)$

* GRADIENT DESCENT (BACKPROP.)

  $dz^{[L]} = \hat{Y} - Y$

  ↳ FROM THIS! CALCULE THE REST OF THE DERIVATIVES.

→ SOME DEEP LEARNING FRAMEWORKS

- CAFFE/CAFFE 2
- CNTK
- DL4J
- KERAS
- LASAGNE
- MXNET
- PADDLEPADDLE
- PYTORCH
- TENSORFLOW
- THEANO

* CHOOSING FRAMEWORKS
  - → EASE OF PROGRAMMING (DEV AND DEPLOYMENT)
  - → RUNNING SPEED
  - → TRULY OPEN (OPEN-SOURCE WITH GOOD GOVERNANCE)