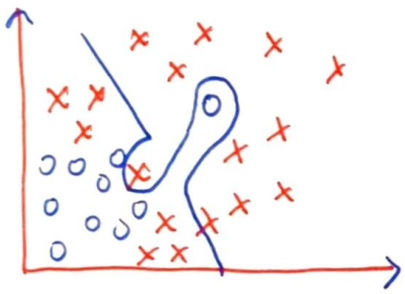
Improving Neural Networks: Hyperparameter Tunning

Biase & Variance

Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	=> High Variance	=> High biase	=> high variance => high biase	=> low biase => low variance

Optimal error (Bayes) error: ≈ 0%



=> High Biase + High Variance

Basic Recipe for Machine Learning:

1st) High Biase? (training data performance)
- Y → • Bigger Network • Time longer (NN architecture search ~~structure~~) until remove biase problem
- N ↓

2nd) High Variance? (dev set performance)
- Y → • More data • Regularization (NN architecture search)
- N ↓

Done

"Biase Variance trade off"
- -> earlier Deep Learning era
- ∵ not much methods to correct one without hurting other
- But now it can be done in Deep Learning

REGULARIZATION

- to prevent overfitting or reduce variance

Logistic regression: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$, λ = regularization parameter

$\min J(w,b)$

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

L_2 regularization: $\|w\|_2^2 = \sum_{j=1}^{n_x} = w^T w$

L_1 Regularization: $\frac{\lambda}{m}\sum_{j=1}^{n_x} |w| = \frac{\lambda}{2m}\|w\|_1$

w will be sparse $\because$ a lot of zeros

$\because$ b is only one parameter, where as w is high dimensional parameter vector, which is more beneficial to be regularized

Neural Network:

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

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2 \quad : \text{"Frobenius norm"}$$

$w: (n^{[l-1]}, n^{[l]})$

$$dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} w^{[l]}$$

"weight decay"

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

$$\Rightarrow w^{[l]} = w^{[l]} - \alpha\left[(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}\right]$$

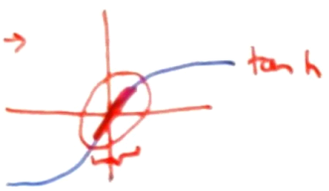
$$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha\,(\text{from backprop})$$

or $$w^{[l]} = w^{[l]}\underbrace{\left(1 - \frac{\alpha\lambda}{m}\right)}_{\text{slightly less than 1}} - \alpha\,(\text{from backprop})$$

How does
~~Why~~ Regularization prevent Overfitting?

→ By reducing weights by smaller & smaller unit, $w^{[l]} \approx 0$, which may turn the problem of high variance or overfitting to a problem of high biase or underfitting.

But in the process, there must be an intermediary point where the fitting is "just right".



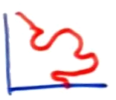
If $\lambda \uparrow \Rightarrow w^{[l]} \downarrow$ ∵ $z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$

∴ if $w^{[l]} \downarrow \Rightarrow z^{[l]} \downarrow\downarrow$

it will be in small range (shown in graph)

∴ $g(z)$ will be linear

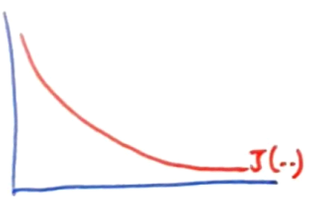
Every layer ≈ linear

preventing it from taking complex form like: 

⇒ Reducing Overfitting

Implementation tip:

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



⇒ Now new J() has this additional term

So while doing gradient discent, it must be included in the function

Dropout Regularization:

Go through each layer of NN and set some probability of dropping some neuron from the Network.

⇒ Removing ingoing and outgoing nodes from it

⇒ Reduced Network ⇒ smaller examples to be trained ⇒ Reducing Overfitting

1) Inverted Dropout:

Illustrate with layer $l = 3$, keep-prob = 0.8

$d3 = np.random.rand(a3.shape[0], a3.shape[1]) \leq keep.prob$

$a3 = np.multiply(a3, d3)$ { chances of elements of d3 to be zero, this makes activations for that elements to be zero

$a3 \;/= keep\text{-}prob$ → Inverted Dropout

Eg.: 50 units in $l = 3$, if 0.8 probability of keeping them ⇒ 10 units shut off

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

↳ reduced by 20%

$/= 0.8$

⇒ Making test time easier ☺

Making predict^ @ test time:

$a^{[0]} = x$

No drop out

$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$

$a^{[1]} = g^{[1]}(z^{[1]})$

$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$

$a^{[2]} = \ldots$

$\downarrow$

$\hat{y}$

Why does drop out work?

One Neuron
^ can't rely on any one feature, so have to spread out weights on each of previous units ⇒ shrinking weights (squared norm of weights)

⇒ ~ L_2 Regularizat^

→ Can have keep-prob diff. for different layer
- layers with like more parameters can have ~~higher~~ lower keep-prop
- layer with less parameter can have keep-prop close to 1.0

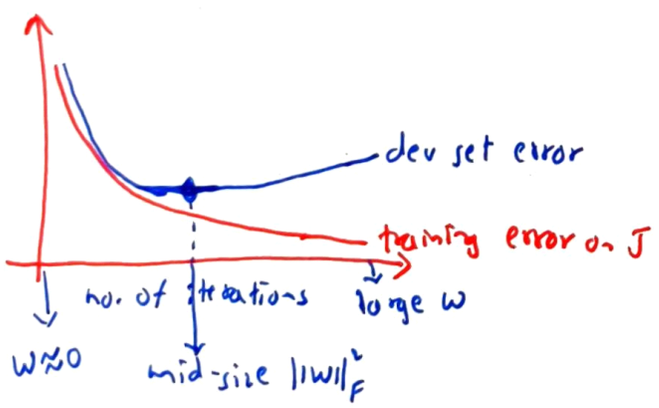
Downside:
- cost f^ J(..) is not well defined

Other Regularization Techniques:

i) Data ~~augmentation~~ augmentation:

Like flipping image ; Random crops of image ; random distortat^ & transformat^

ii) Early stopping:

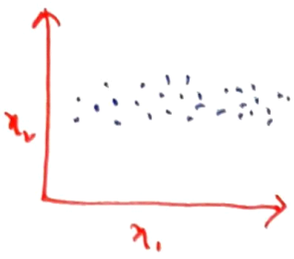


Downside:

In ML:
- optimize cost f^ J } Orthogonalizat^
- Not overfit

- Early stopping both these tasks and can't do one at a time, so can't try much complicated tasks

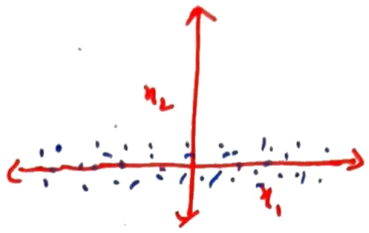
Normalizing Inputs



Step I: Subtract Mean

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

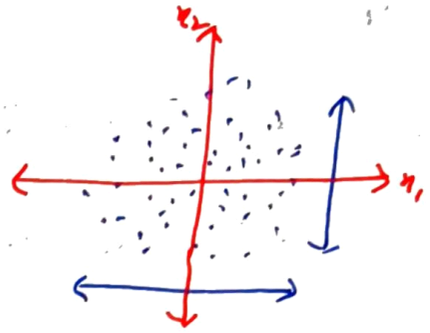
$$x_i = x - \mu$$



Step II: Normalize Variance:

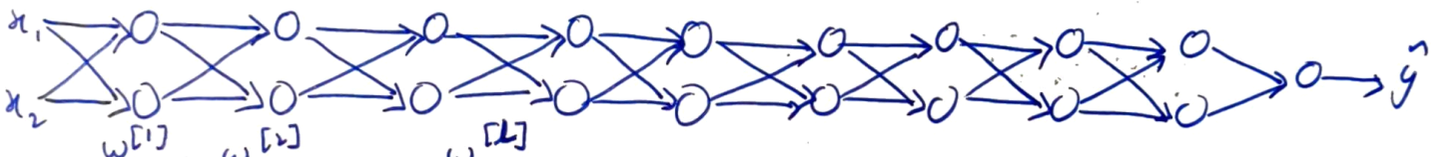
$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} ** 2 \quad \hookrightarrow \text{element-wise square}$$

$$x /= \sigma^2$$



Note: Use same μ & σ^2 to normalize train & test set

VANISHING / EXPLODING GRADIENTS



$w^{[1]}, w^{[2]}, \dots w^{[L]}$

$g(z) = z \quad , \quad b^{[l]} = 0$

$$\therefore \quad \hat{y} = w^{[L]} w^{[L-1]} w^{[L-2]} \dots w^{[3]} w^{[2]} \underbrace{w^{[1]} x}_{z^{[1]}}$$

$$z^{[1]} = w^{[1]} x$$

$$\underbrace{a^{[1]}}_{z^{[1]}} = g(z^{[1]}) = z^{[1]}$$

$$a^{[2]} = g(z^{[2]}) = g(w^{[2]} a^{[1]})$$

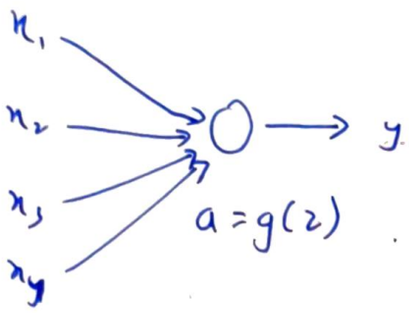
$$w^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \quad \hat{y} = w^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} x \rightarrow 1.5^{L-1} x$$

$$\text{if } w^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \quad \dots \quad \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}^{L-1} x \rightarrow 0.5^{L-1} x$$

$\Rightarrow$ $w^{[l]} > I$ $\Rightarrow$ Activations can explode

if $w^{[l]} < I$ $\Rightarrow$ Activations can decrease exponentially

Weight Initialization for Deep Networks



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

let, $b = 0$

larger the n is smaller w_i

one way: $Var(w_i) = \frac{1}{n}$

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

If ReLU: $Var(w_i) = \frac{2}{n}$, $g^{[l]}(z) = ReLU(z)$

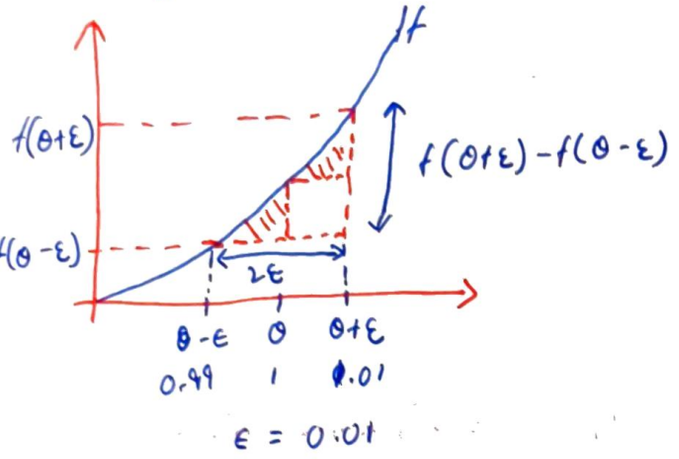
$$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$$

If tanh: $var(w_i) = \sqrt{\frac{1}{n^{[l-1]}}}$ } - Xavier Initialization

$$\sqrt{\frac{2}{n^{[l-1]} * n^{[l]}}}$$

Numerical approximation of gradients

Checking your derivative computation:



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

$$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001$$

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

approx error : 0.0001

(pre slide : 3.0301 error : 0.03)

$$\because f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta+\varepsilon)}{2\varepsilon} \to O(\varepsilon^2)$$

$$= (0.01)^2 = 0.0001$$

GRADIENT CHECKING

Take $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \ldots, w^{[L]}, b^{[L]}$ & reshape into a big vector θ

(Concatinate)

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

Nly do with $dw^{[1]}, db^{[1]}, \ldots, dw^{[L]}, db^{[L]}$ & reshape into big vector $d\theta$

(Concatinate)

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

⇒ Gradient checking (Grad Check)

~~for eg~~

Grad Check:

for each i:

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

$$\approx d\theta^{[i]} = \frac{\partial J}{\partial \theta_i}$$

Check if

$d\theta_{approx} \approx d\theta$

To do so:

check $\dfrac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$ → Euclidian distance

In practice: $\varepsilon = 10^{-7}$; then → $\approx 10^{-7}$ - great

$\approx 10^{-5}$ - need to be checked

$\approx 10^{-3}$ - worry

Implementat° of Grad Check:

- Don't use in training - only to debug
- If algo fails grad check, look @ compo-nts to try to identify bug. try to find how where they arise $\left\langle \begin{matrix} dw \\ db \end{matrix} \right.$
- Remember Regularizat°

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

$d\theta$ = gradient of J wrt. θ [includes regularizat° term]

- Doesn't work with dropout
  - to ~~work with~~ check - turn off dropout - keep-prob = 1.0
- Run @ random initializat°; perhaps again after some training

Optimization Algorithms

Mini-Batch Gradient Descent:

Vectorization allows you to efficiently compute on m examples.

$$X = [x^{(1)}, x^{(2)}, x^{(3)} \ldots x^{(m)}] \quad (n_x, m)$$

$$Y = [y^{(1)}, y^{(2)}, y^{(3)} \ldots y^{(m)}] \quad (1, m)$$

What if $m = 5{,}000{,}000$? - training still be very slow

∴ Mini-batches of 1,000 each

$$X = [\underbrace{x^{(1)}\ x^{(2)} \ldots x^{(1{,}000)}}_{X^{\{1\}}_{(n_x, 1000)}}\ \underbrace{x^{(1{,}001)} \ldots x^{(2{,}000)}}_{X^{\{2\}}_{(n_x, 1000)}} \ldots \underbrace{\ldots x^{(m)}}_{X^{\{5{,}000\}}_{(n_x, 1000)}}]$$

$$Y = [\underbrace{y^{(1)}\ y^{(1)} \ldots y^{(1{,}000)}}_{Y^{\{1\}}_{(1, 1000)}}\ \underbrace{y^{(1{,}001)} \ldots y^{(2{,}000)}}_{Y^{\{2\}}_{(1, 1000)}} \ldots \underbrace{\ldots y^{(m)}}_{Y^{\{5{,}000\}}_{(1, 1000)}}]$$

For $t = 1, \ldots 5{,}000$ → 1 step of gradient descent using $\{x^{\{t\}}, y^{\{t\}}\}$

Forward prop on $X^{\{t\}}$: } Vectorized implementation (1,000 example)

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

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 * 1000} \sum_{l} \|w^{[l]}\|_F^2$

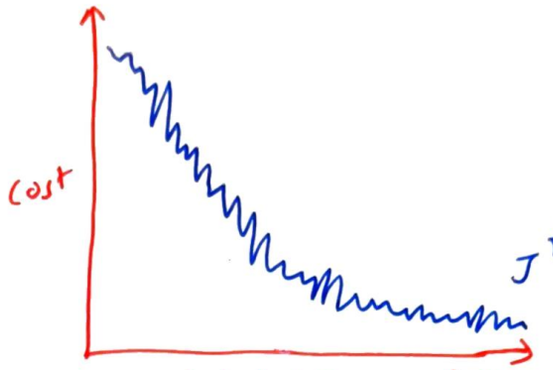
Backprop to compute gradients using $J^{\{t\}}, x^{\{t\}}, y^{\{t\}}$

$$w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}; \quad b^{[l]} = b^{[l]} - \alpha\, db^{[l]}$$

"1 epoch" → pass through training sets

Training with mini batch gradient descent!

Batch Gradient Descent



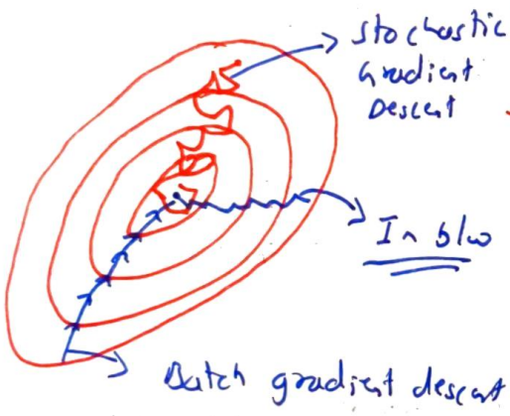
Plot $J^{\{t\}}$ computed using $x^{\{t\}}, y^{\{t\}}$

Mini Batch Gradient descent

Size of mini-batch:

- If mini batch size = m : Batch gradient descent $(x^{\{1\}}, y^{\{1\}}) = (x, y)$

If mini batch size = 1 : Stochastic gradient descent - every eg. is own mini batch



Lose speedup from vectorizat^n

In practice: somewhere in blw 1 & m

Fastest Learning
- vectorizat^n
- Make progress without needing to wait for processing entire training set

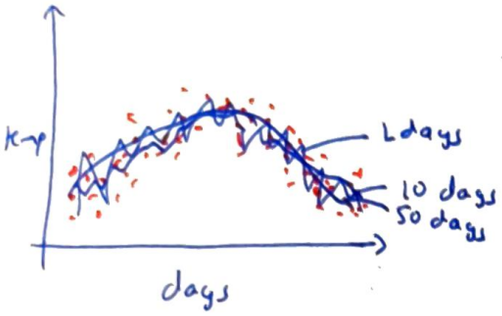
Note: - If small train^g set: - Use batch gradient descent ($m \leq 2000$)

- Typical mini-batch sizes: 64, 128, 256, 512 (2^6, 2^7, 2^8, 2^9)

- ~~Mini ba~~ Make sure mini-batch fits in CPU/GPU memory $x^{\{t\}}, y^{\{t\}}$

Exponentially Weighted Averages:

Temperature in London:



$V_0 = 0$

$V_1 = 0.9 V_0 + 0.1 \theta_1$

$V_2 = 0.9 V_1 + 0.1 \theta_2$

$V_3 = 0.9 V_2 + 0.1 \theta_3$

$\vdots$

$V_t = 0.9 V_{t-1} + 0.1 \theta_t$

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

V_t: approximately average over $\approx \frac{1}{1-\beta}$ days' temp.

$\beta = 0.9 \quad \approx$ 10 days' temp

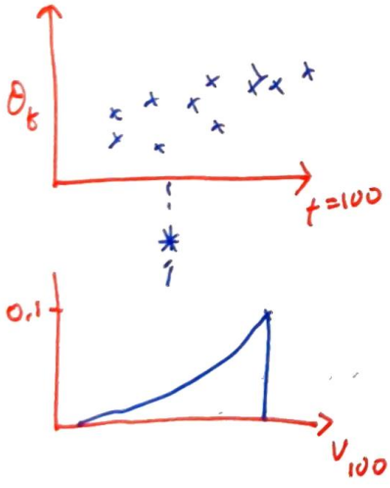
$\beta = 0.98 \quad \approx$ 50 days

$\beta = 0.5 \quad \approx$ 2 days

$V_{100} = 0.99 V_{99} + 0.1 \theta_{100}$ $\rightarrow 0.1\theta_{98} + 0.9 V_{97}$

$V_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9 V_{98})$

$= 0.1\theta_{100} + 0.1 \times 0.9 \theta_{99} + 0.1 (0.9)^2 \theta_{98} + 0.1 (0.9)^3 \theta_{97} + \cdots$



$0.9^{10} \approx 0.35 \approx \frac{1}{e}$

$0.98^{50} \approx \frac{1}{e}$

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

{ $\varepsilon = 0.1$ in this case

$\varepsilon = 1 - \beta$

Implementing exponentially weighted avg.:

$V_0 = 0$

$V_1 = \beta V_0 + (1-\beta)\theta_1$

$V_2 = \beta V_1 + (1-\beta)\theta_2$

$\vdots$

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

$V_\theta = 0$

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

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

$\vdots$

$V_\theta = 0$

Repeat {

  Get Next θ_t

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

}

BIAS CORRECTION:

$V_t = \beta V_{t-1} + (1-\beta)\theta_t$, Let $\beta = 0.98$

$V_0 = 0$

$V_1 = 0.98 V_0 + 0.02\theta_1 = 0.02\theta_1$

$V_2 = 0.98 V_1 + 0.02\theta_2 = 0.98(0.02)\theta_1 + 0.02\theta_2$

$= 0.0196\theta_1 + 0.02\theta_2$

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

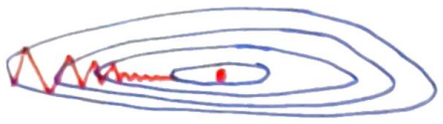
When $t = 2$: $1 - \beta^t = 1 - (0.98)^2 = 0.0396$

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

⇒ Removes biose

⇒ as t increases the biose etc. approaches 0 ($\because \theta$ approaches 0)

Gradient Descent with Momentum



i) It prevents from using higher learning rate, as when used, gradient descent will overshoot

ii) Want ~~test~~ slower learning on vertical axis
But want faster learning on horizontal axis

=> Gradient descent with momentum:

Momentum:

On iteration t:

Compute dw, db on current mini-batch

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

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

Update weights:

$$w = w - \alpha V_{dw} \quad ; \quad b = b - \alpha V_{db}$$

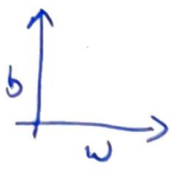
What it does: is Average out the oscilation on both axis =>
- approx. 0 on vertical axis
- bigger on horizontal axis

=> 2 Hyperparameter α (learning rate), β (Momentum)

standard $\beta = 0.9$

$\approx$ avg. over last ≈ 10 gradients

RMSprop (Root Mean Square Prop):



- want to slow down learning on b axis
- " " fast the " " w axis

On iteration t:

Compute dw, db on current mini-batch

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2 \quad \text{(element wise square; hoping small)}$$

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

$$w := w - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon} \quad \text{(small)} \qquad b := b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon} \quad \text{(large)}$$

$\Rightarrow$ w updated fast

$\Rightarrow$ b update will be slow

Adam Optimization Algorithm:

(Momentum + RMSprop)

$V_{dw} = 0, \; S_{dw} = 0, \; V_{db} = 0, \; S_{db} = 0$

On iteration t:

Compute dw, db using current mini-batch

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw, \quad V_{db} = \beta_1 V_{db} + (1-\beta_1) db \leftarrow \text{momentum}$$

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2, \quad S_{db} = \beta_2 S_{db} + (1-\beta_2) db \leftarrow \text{RMS}$$

$$V_{dw}^{\text{corrected}} = \frac{V_{dw}}{(1-\beta_1^t)}, \quad V_{db}^{\text{corrected}} = \frac{V_{db}}{(1-\beta_1^t)}$$

$$S_{dw}^{\text{corrected}} = \frac{S_{dw}}{(1-\beta_2^t)}, \quad S_{db}^{\text{corrected}} = \frac{S_{db}}{(1-\beta_2^t)}$$

Updating:

$$w := w - \alpha \frac{V_{dw}^{\text{corrected}}}{\sqrt{S_{dw}^{\text{corrected}}} + \varepsilon} \qquad b := b - \alpha \frac{V_{db}^{\text{corrected}}}{\sqrt{S_{db}^{\text{corrected}}} + \varepsilon}$$

Hyperparameters choices:

α : needs to be tune

β_1 : 0.9 (dw)

β_2 : 0.999 (dw^2)

ε : 10^{-8}

Usually - ~~fix~~ use std. values of β_1, β_2 & ε & try tuning α

Learning Rate Decay:

1 epoch = 1 pass through the data

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

$\alpha_0 = 0.2$

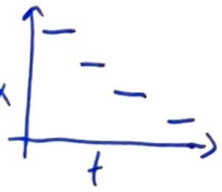
decay rate = 1

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
:	:

Other learning rate decay methods:

i) $\alpha = 0.95^{\text{epoch-no.}} \cdot \alpha_0$ - exponential decay

ii) $\alpha = \frac{k}{\sqrt{\text{epoch no.}}} \cdot \alpha_0$

iii) 

→ discrete decay

iv) Manual decay

HYPERPARAMETERS TUNING

α → most imp.

$\beta \sim 0.9$

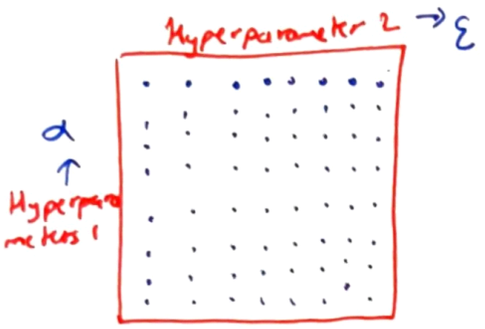
β_1, β_2, ε

no. of layers

no. of hidden units. — 2nd most imp.

learning rate decay

mini-batch size



Grid Method

↓

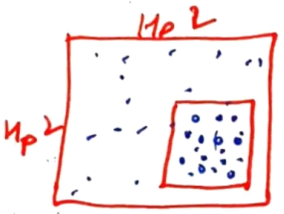
[useful only when very few hyperparameters]

Random

↓

[Better for more no. hyperparameters]

Coarse to fine scheme →



- start by exploring bigger region
- then try focusing on smaller region where you are getting better results
- then do more sampling in that region

Using an appropriate scale to pick hyperparameters:

i) Picking hyperparameters @ random:

$n^{[l]} = 50, \ldots, 100$

50 ——— 100 } picking no. @ random

no. of layers : 2 - 4 } Sampling layer @ random

ii) Appropriate scale for hyperparameters:

$\alpha = 0.0001, \ldots, 1$

0.0001 ——— 1 {* Linear scale not to be used}

0.0001 0.001 0.01 0.1 1 } Logarithmic scale ✓

$r = -4 * np.random.rand()$ ← $r \in [-4, 0]$

$\alpha = 10^r$ ← $10^{-4} \_\_ 10^0$

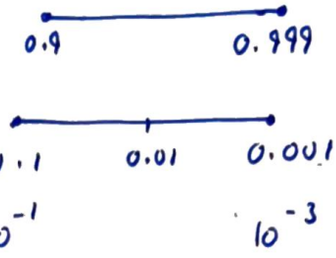
10^a ... 10^b $\quad r \in [a, b]$ $\quad \alpha = 10^r$

In this eg. $a = \log_{10}(0.0001) = -4$, $b = \log_{10}(1) = 0$

iii) Hyperparameters for exponentially weighted averages:

$\beta = 0.9$ ---- 0.999

↳ $1 - \beta = 0.1$ ---- 0.001



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

10^{-1} $\quad$ 10^{-3}

$1 - \beta = 10^r$

$\beta = 1 - 10^r$

NOTE:

If β is close to 1, the sensitivity of results gets changes even with very small changes in β

If β : $0.9000 \rightarrow 0.9005$ → might not have much effect on the changes

β : $0.999 \rightarrow 0.9995$ → will have a greater effect

} ~ 10 values

∴ Logarithmic scale should be used ~~instead~~ as it give proper weightage to the scale, rather than linear scale

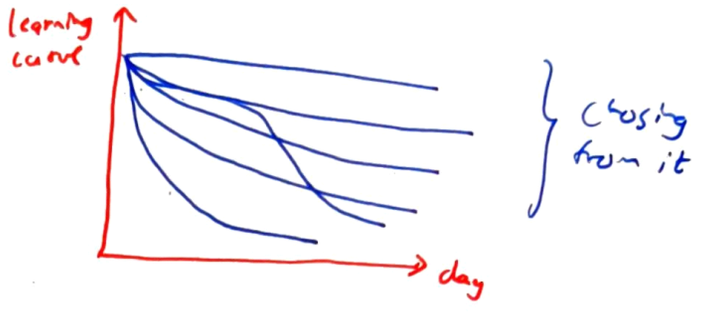
Hyperparameters tuning in practice: Pandas Vs. Caviar

1) Babysitting One Model:

Panda Approach

{∵ the have very few children, but when have, take care of them only}

2) Training Many models in parallel:



Caviar Approach

{∵ when fish reproduce, the reproduced many eggs and see if any of them survive}

BATCH NORMALIZATION

Normalizing inputs to speed up learning

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

~~$X = X - \mu$~~

$$\sigma^2 = \frac{1}{m} \sum_i x^{(i)2}$$

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

In deeper Nets: Can we normalise the values of $a^{[l]}$ so as to train $w^{[l+1]}$, $b^{[l+1]}$ faster ?

Normalize $z^{[l]}$ or $A^{[l]}$

↓

more used in practice

Given some intermediate values in NN $z^{(1)}, \dots, z^{(m)}$ $\quad (= z^{[l](i)})$

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

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

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

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

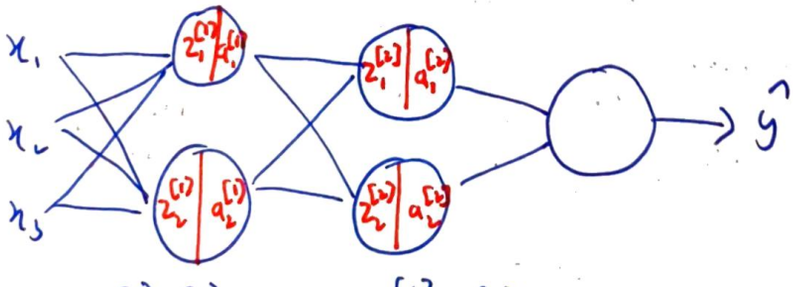
γ, β: Learnable parameters of model

$\Rightarrow$ allows to set mean of $\tilde{z}^{(i)}$ whatever you want it to be

If: $\gamma = \sqrt{\sigma^2 + \varepsilon}$, $\beta = \mu$ Then: $\tilde{z}^{(i)} = z^{(i)}$

Now will use $\tilde{z}^{(i)}$ instead of $z^{[l](i)}$

Adding Batch Norm to a Network:



$$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]}$$

$$z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \rightarrow a^{[2]}$$

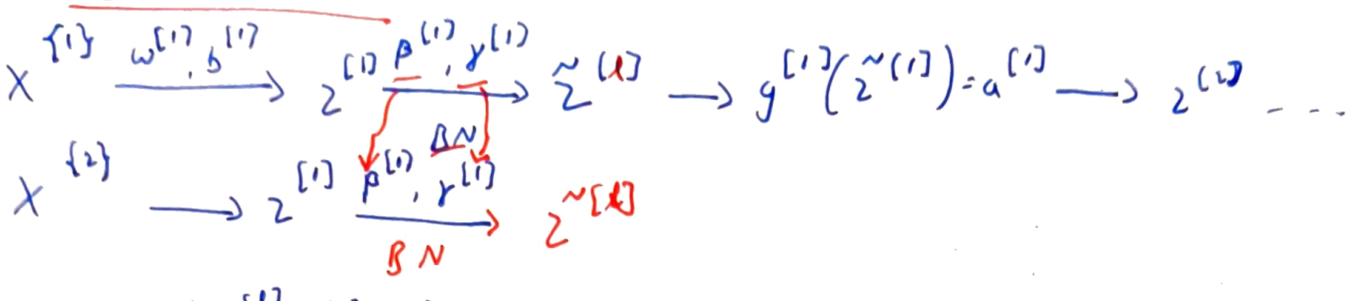
Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots, w^{[L]}, b^{[L]}$

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

$d\beta^{[l]}$

$$\beta^{[l]} = \beta^{[l]} - d\beta^{[l]}$$

Working with mini-batch:



Params: $w^{[l]}$, $b^{[l]}$, $\beta^{[l]}$, $\gamma^{[l]}$

$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$

⇒ Batch Norm 1st scale $z^{[l]}$ to mean 0 and then rescale it w.r.t β & γ

⇒ whatever is the value of $b^{[l]}$, it will get subtracted out ∴ $b^{[l]}$ is ignored

Mean Subtract step

$$\Rightarrow z^{[l]} = w^{[l]} a^{[l-1]}$$

$z^{[l]}_{norm}$

$$\tilde{z}^{[l]} = \gamma^{[l]} z^{[l]}_{norm} + \beta^{[l]}$$

Implementing gradient descent:

for $t = 1 \ldots$ no. of mini batches

Compute forward prop on mini batch $X^{\{t\}}$

In each hidden layer, use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$

Use back prop to compute $dw^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$

{no. db, ∵ we removed b

Update parameters $w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$

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

$\gamma^{[l]}$ ....

Why Batch Norm Works?

- It makes weights deeper in ~~deeper~~ NN more robust to changes as compared to earlier weights in NN

"Covariate Shift" - If we maped $X \rightarrow y$ & trained the model, & if the distribution of X changes we have to retrain the model
- It is true even if the ~~m~~ function does not changes
- It become worse if function also changes

Happens in DNN when $a^{[l]}$ (or $z^{[l]}$) of one layer keeps on changing because of changing $w^{[l-1]}, b^{[l-1]}$ and this in turn affect the relatn with $w^{[l+1]}, b^{[l+1]}$

- But even if the values of $z^{[l]}$ changes, but the value of mean & variance remain same, thus $\beta^{[l]}, \gamma^{[l]}$

=> allowing one layer to learn somewhat independently from previous layers

2) Batch Norm as regularizatn:

- Each mini-batch is scaled by the mean & variance computed on just that mini-batch => little bit noise to $z^{[l]}$

=> making scaling process of $z^{[l]}$ noisier

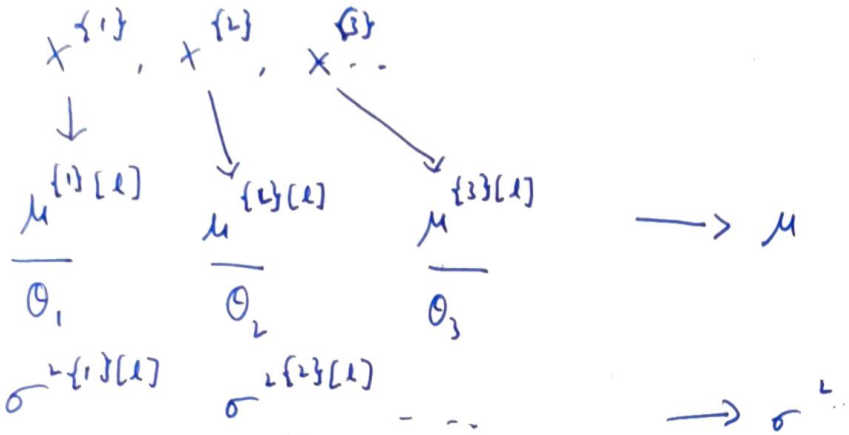
- So ~~N~~ to dropout, it adds some noise to each hidden layer's activations.

=> Slight regularizatn effect ∵ noise added is small

If use a bigger mini-batch size -> reduces this regularizatn effect

Batch Norm at test time:

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



At test time:

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

MULTIPLE-CLASS CLASSIFICATION

SOFTMAX: on layer L ; classes $= 0, 1, 2, 3$

$$z^{[L]} = w^{[L]} a^{[L-1]} + b^{[L]} \qquad (4,1)$$

Activation:

$$t = e^{(z^{[L]})}$$

$$a^{[L]}_{(4,1)} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{4} t_i} \quad ; \quad a^{[L]}_i = \frac{t_i}{\sum_{j=1}^{4} t_i}$$

Let, $z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$; $t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$. $\sum_{j=1}^{4} t_j = 176.3$

$$a^{[L]} = \frac{t}{176.3} = \frac{1}{176.3}\begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

$$a^{[L]}_{(4,1)} = g^{[L]}\left(z^{[L]}_{(4,1)}\right)$$

Training a Softmax classifier:

$z^{[L]}_{(4,1)} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$ $t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$ $C = 4$, $g^{[L]}$ $a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$

Contrast to "hard max"

$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

Softmax regression generalizes logistic regression to C classes.

Loss function:

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \text{ - cat} \qquad a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \qquad (= 4$$

$$\boxed{\mathcal{L}(\hat{y}, y) = -\sum_{j=1} y_j \log \hat{y}_j}$$

In given eg. $y_1 = y_3 = y_4 = 0$

$\Rightarrow \mathcal{L} = -y_2 \log \hat{y}_2 = -\log \hat{y}_2$ $\}$ => need to make $\hat{y}_2$ big

Cost:

$$\boxed{J(w^{[1]}, b^{[1]}, \dots) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}$$

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \cdots \\ 1 & 0 & 0 & \\ 0 & 0 & 1 & \\ 0 & 1 & 0 & \end{bmatrix}_{(4,m)}$$

$$\hat{Y} = \begin{bmatrix} \hat{y}^{(1)}, & \cdots, & \hat{y}^{(m)} \end{bmatrix} = \begin{bmatrix} 0.3 & \vdots & \\ 0.2 & \vdots & \cdots \\ 0.1 & \vdots & \\ 0.4 & \vdots & \end{bmatrix}_{(4,m)}$$

Gradient Descent with Softmax:

$$z^{[L]}_{(4,1)} \rightarrow a^{[L]} = \hat{y} \rightarrow \mathcal{L}(\hat{y}, y)$$

Backprop: $dz^{[L]}_{(4,1)} = \hat{y} - y$

$\rightarrow \dfrac{\partial J}{\partial z^{[L]}}$