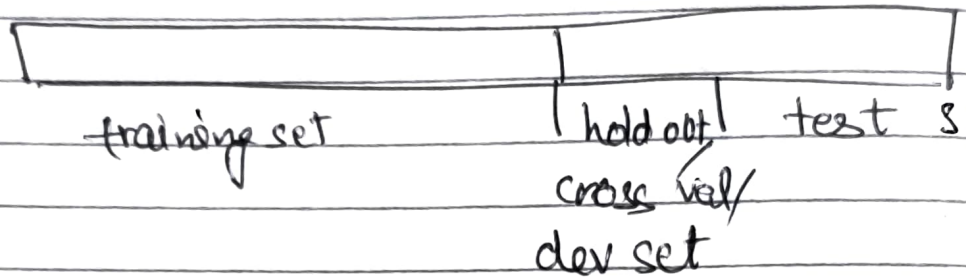
Improving Deep Neural Networks

WEEK 1

#Train/dev/test sets



for 100 - 1000 - 10000 values
Split: 70/30 or 60/20/20

But in big data era with a million values
test data percentage can
be reduced
like 98/1/1

Mismatched train test distr.

Training set	dev/test sets
HD pictures from Web of cat	Cat pictures from your camera/app

make sure dev and test set come from
the same distribution

5:45 $\|w^{[l]}\|^2 = \sum_{i=1}^{n^l} \sum_{j=1}^{n^l} (w_{ij}^{[l]})^2$

Bias / Variance

train set error 1% } high variance
dev set error 11% }

15% } high bias
16% }

15% } high variance
30% } high bias

Basic Recipe

High Bias (Training data problems) → bigger ~~net~~ neural net, more layers

High variance (dev set performance) → More train data
Regularization
more appropriate NN arch

Regularization

for Logistic-

$\min_{w,b} J(w,b) \quad w \in \mathbb{R}^{n_x}, \; b \in \mathbb{R}$

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

L2 regularization

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

L1 regularization

$$\frac{\lambda}{2m} \sum_{i=1}^{m_x} |w| = \frac{\lambda}{2m} ||w||_1$$

w will be sparse

λ is the regularization parameter

* Neural Network

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

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

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

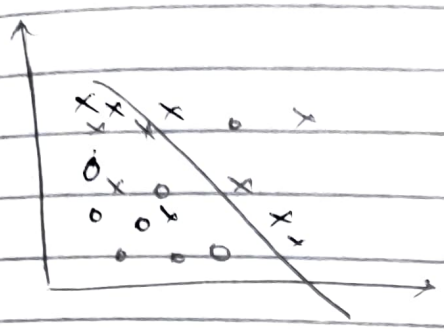
Frobenius norm

weight updates

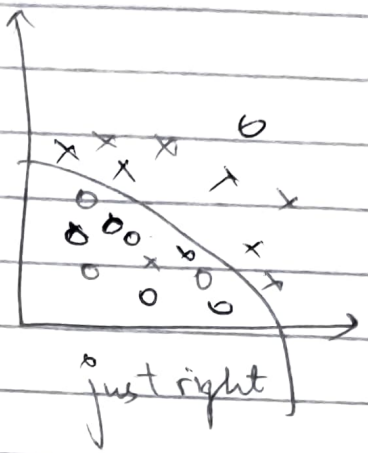
$$w^{[l]} = w^{[l]} - \alpha [(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}]$$

also called weight decay

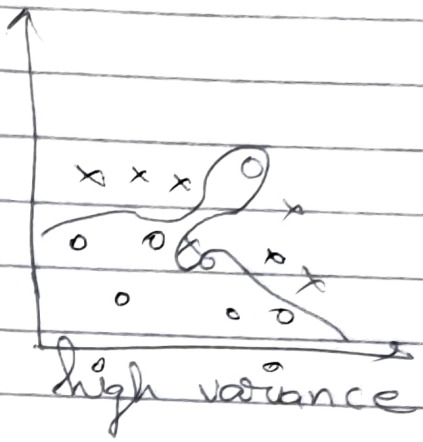
$$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha (\text{from backprop})$$



high bias



just right



high variance

- build better model

- get more data
- reduce model complexity

regularization also helps getting a more monotonically decreasing cost fn curve

Dropout Regularization

* Inverted Dropout

L = 3 (layers)

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

↳ creates a boolean matrix with probability of False = 0.2

a3 = np.multiply(a3, d3)

↳ 20% of the values will be made 0 aprox

a3 /= keep_prob
↳ 0.8

dropout is usually used to stop NN from overfitting

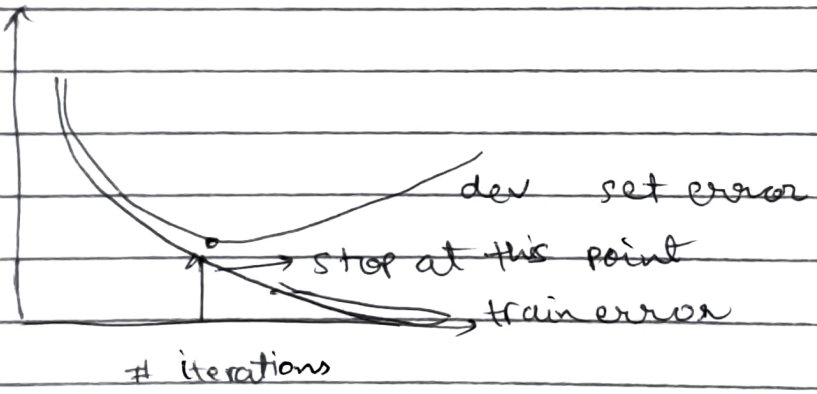
$w^{[1]} : 7 \times 3$ $w^{[3]} : 3 \times 7$

PAGE NO.:
DATE: / /

- Intuition: Cant rely on any one feature, so have to spread out weights.

We increase the dropout usually for higher no of input neurons for that layer

More regularization techniques
- Data augmentation: ex for a CV project you can flip images, rotate them, crop them
- Early Stopping:



Orthogonalization: we have two tasks
- optimize w, b to get minimum J
- set up regularizat^n hyperparameters to stop overfitting

Setting up ~~Normali~~ optimization problem

* Normalizing training sets

mean $x := x - \mu$

var $x := x / \sigma$

* Vanishing / Exploding Gradients

activation values & exponentially increase/decrease

When there are many layers. Use proper weight initializations

* Weight Initializations for DNN.

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

large $n \rightarrow$ small w_i

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

ReLU has other variations:

$\sqrt{\left(\frac{1}{n^{[l-1]}}\right)}$, $\sqrt{\left(\frac{2}{n^{[l-1]} + n^{[l]}}\right)}$

* Gradient Checking

Optimization Algorithms

* Mini Batch:

We have vectorized inputs

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

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

What if m = 5000 000

divide it into min batches

lets say of 1000 training examples each

$$X = [\, \underbrace{x^{(1)} \; x^{(2)} \; \cdots \; x^{(1000)}}_{X^{\{1\}}} \; \underbrace{x^{(1001)} \; x^{(1002)} \; \cdots \; x^{(2000)}}_{X^{\{2\}}} \; \cdots \; X^{\{5000\}} \,]$$

$X^{\{1\}}$: $(n_x, 1000)$

$$Y = [\, y^{(1)} \; y^{(2)} \; \cdots \; y^{(1000)} \; \cdots \,]$$

@ $X^{\{1\}}$

We usually shuffle our X & Y before partioning

Algo.

Repeat epochs {

for t=1 to t=5000 {

forward prop on $X^{\{t\}}$

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

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

⋮

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

} vectorized 1000 examples

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

$+ \frac{\lambda}{2 \cdot 1000} \sum_{l} \|w^{[l]}\|_F^2$

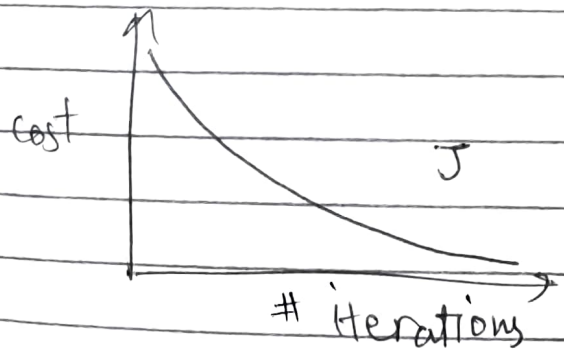
Backprop to compute gradients ...

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

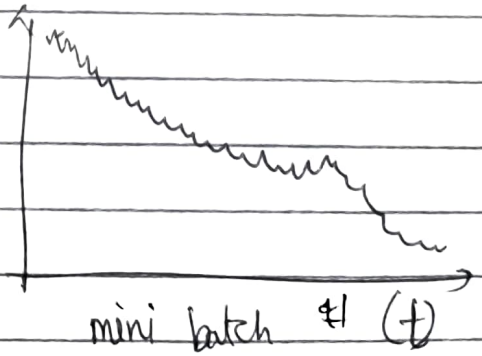
}

}

Batch gradient descent



Mini Batch GD



If mini batch size (~~size~~) = m
: we have batch gradient descent
ie $(x^{\{1\}}, y^{\{1\}}) = (x, y)$

If mini batch size = 1
: we have Stochastic GD
every example is its own mini batch

Stochastic	in between	batch
size = 1	= m	
mbsize = 1	mb size not too	mbsize = m
↓	big not too small	↓
lose speed from	↓	too long per
vectorization	fastest learning	iteration
	~~Optimal~~ ↓	for large m

If small m : use batch GD
$m \leq 2000$
Typical mini batch sizes : 64, 128, 256, 512 ..

* Exponentially weighted Averages

Moving averages are usually seen in time series datasets.

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

$\beta = 0.9 \Rightarrow 1/1-0.9 \approx$ averaging over last 10 days
$\beta = 0.98 \Rightarrow 1/1-0.98 \Rightarrow$ averaging over last 50 days
↳ curve will be smother, less sensitive to immediate changes

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

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

$V_{98} = 0.9 V_{97} + 0.1 \theta_{97}$

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

(V_{99}, V_{98})

$= 0.1\theta_{100} + 0.1 \times (0.9)^1 \theta_{99} + 0.1(0.9)^2 \theta_{98} + 0.1(0.9)^3 \theta_{97} \cdots$

implementation:

$V := 0$

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

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

$\vdots$

$V_0 = 0$

repeat {

get θ_t

$V = \beta V + (1-\beta)\theta_t$

}

* bias correction

your predictions lean towards 0 initially

when you start with $v=0$

* GD with momentum: to avoid oscillations in GD

$V_{dw} = V_{db} = 0$ on iteration t:

compute dw, db on current mini batch

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

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

$w := w - \alpha V_{dw}$, $b := b - \alpha V_{db}$

$\beta = 0$ implies GD w/o momentum

* RMSprop

On iteration t

compute dw, db on current mini batch

$S_{dw} = \beta S_{dw} + (1-\beta) dw^2$ ← small

$S_{db} = \beta S_{db} + (1-\beta) db^2$ ← large

$w := w - \alpha \frac{dw}{\sqrt{S_{dw}}}$ $\quad b := b - \alpha \frac{db}{\sqrt{S_{db}}}$

* Adam optimization Algo

$V_{dw} = 0$ $\quad S_{dw} = 0$ $\quad V_{db} = 0$ $\quad S_{db} = 0$

On iteration t

compute dw, db using current mini batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$

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

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

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

$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t)$

dw

$= V_{db} / (1-\beta_1^t)$

$S_{dw}^{corrected} = S_{dw} / (1-\beta_2^t)$

$S_{db} / (1-\beta_2^t)$

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

ε = small term usually 10^{-8}

usually,
α needs to be tuned
$\beta_1 = 0.9$
$\beta_2 = 0.999$
$\varepsilon = 10^{-8}$
Adam: Adaptive moment estimation

* Learning Rate Decay
to tune the α hyperparameter, we use this algorithm
1 epoch = 1 pass through training data

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

$\alpha_0 = 0.2$
decay rate = 1

epoch	α
1	0.1
2	0.067
3	0.05
4	0.04

other methods:

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

$$\alpha = \frac{k}{\sqrt{\text{epoch num}}} \cdot \alpha_0$$

* problem of local optima
  - typical optimas in DL lie at saddle points
  - unlikely to stuck in bad local optima
  - pleatuers can make learning slow
    ↳ use momentum/rmsprop

Practice —

$\theta_1 = 10$

$\theta_2 = 10$

$\beta = 0.5$

~~$v_0 = 0$~~

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

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

$= 0.5 \times 10 = 5$

$V_2 = 0.5 \times 5 + 0.5 \times 10$

$= 7.5$

Hyperparameter Tuning

usually in order of importance of tuning:

- α
- β (momentum)
- #hidden units
- mini batch size
- #layers
- learning rate decay
- $\beta_1, \beta_2, \varepsilon$ (Adam parameters 0.9, 0.999, 10^{-8} work for most of the cases)

for Use appropriate scale

linear: x x x x x x x x x x x x x x x
0 0.1 0.2 0.3 ... 1
90% values lie b/w 0.1 & 1

x: uniform random dist over linear scale

logarithmic: x x x x x x x x x x x x x x x
0.0001 0.001 0.01 0.1 1
10% values lie b/w 0.1 & 1

how to apply:

```
r = -4 * np.random.rand()
α = 10 ** r
```

↳ usually used for tuning α

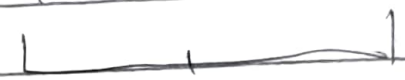
Hyperparameters for exponentially weighted averages

$\beta = 0.9 \quad \ldots \quad 0.999$

0.9 ↓ last 10 values

0.999 ↓ last 1000 values

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



this is because algos are highly sensitive when $\beta \approx 1$

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

$1-\beta = 10^r$

$\beta = 1 - 10^r$

ie $0.999^{\frac{1}{1000}}$ ⇒ last 1000 days

0.9995 → last 2000 days

0.9 → last 10

0.9005 → last 10.5

notice the difference when β is closer to 1

Normalizing Activations Batches

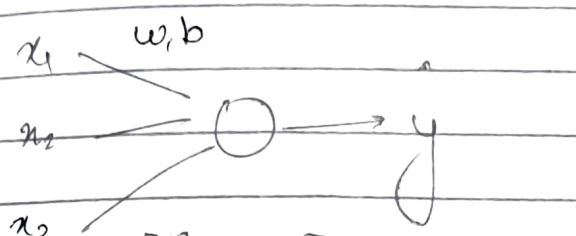
In normalization, we

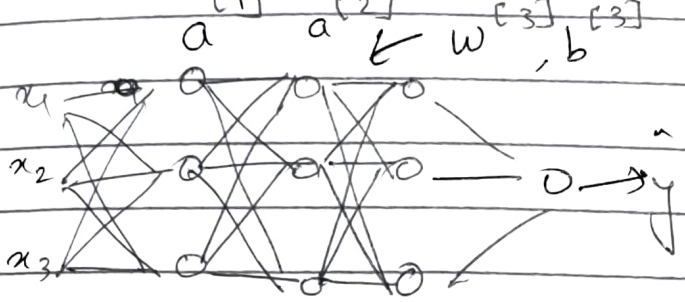
$$\mu = \frac{1}{m}\sum_i x^{(i)}$$

$$x = x - \mu$$

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

$$x = x/\sigma^2$$





we can normalize the inputs to hidden layers too

$z^{[l](i)}$

usually z

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

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

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

If $\gamma = \sqrt{\sigma^2 + \varepsilon}$

$\beta = \mu$

$\tilde{z}^{(i)} = z^{(i)}$

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

Use $\tilde{z}^{(i)}$ instead of $z^{(i)}$

4:30 $a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^{4} t_i}$

Adding Batch normalization to NN

$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{batch norm}]{} \tilde{z}^{[1]} \xrightarrow{} a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \cdots$

parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]} \cdots w^{[L]}, b^{[L]}$

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

Working w mini batches

$X^{\{1\}} \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[BN_1]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \cdots$

only of the batch

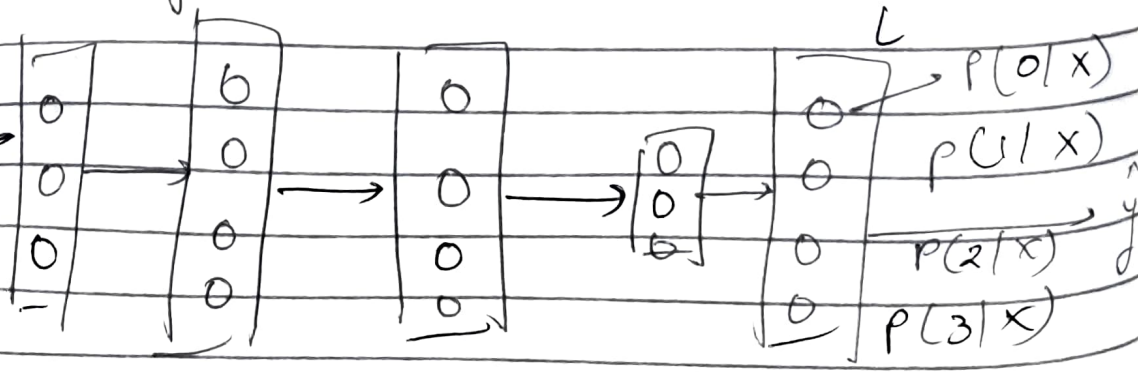
:

Multiclass Classification

- Softmax Regression

Classes: 1 → cat
2 → dog
3 → chicken
0 → none

C = no of classes = 4 (0, 1, 2, 3)



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

Activation fn

$$t = e^{z^{[l]}}$$

$$a^{[l]}_{(4,1)} = \frac{t}{\sum_{i=1}^{4} t_i}, \quad a_i^{[l]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$

$$z^{[l]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \Rightarrow t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$$

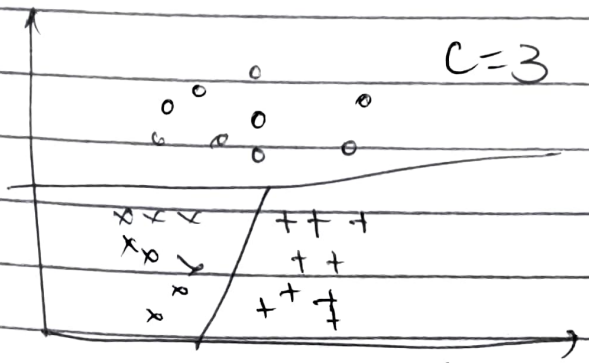
$$a^{[l]} = \frac{t}{176.3}$$

$$\sum_{i=1}^{4} t_i = 176.3$$

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ - \end{bmatrix} \rightarrow \begin{cases} e^5/176.3 = 0.842 \\ e^2/176.3 = 0.042 \end{cases}$$

$$a^{[l]}_{(4,1)} = g^{[l]}(z^{[l]}) \rightarrow (4,1)$$

Ex: $C = 3$



Softmax creates linear boundaries (straight lines)

Softmax generalizes logistic to C classes

if $C = 2$: logistic = softmax