Course 2:

Week 1 Train/dev/Test Sets

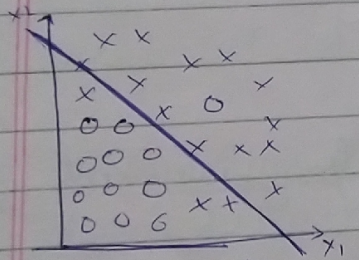
→ Development set is oftenly used to test the trained model!
→ For example. If we have two models and we need to determine which will work better, we test on dev set
→ the dev and test sets must come from the same distribution

Bias/Variance

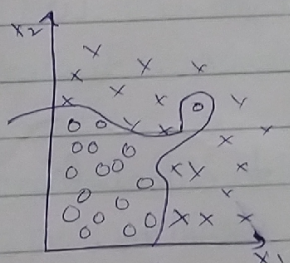
Train set error:	1%	15% cannot fit the training set
Dev set error:	11%	16% as well as dev set
	in this case, high variance	high bias
	overfitting	underfitting

Assuming optimal error is 0%.

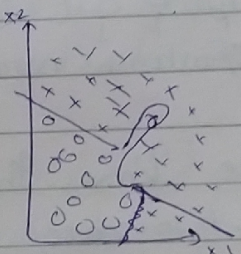
Train set error:	15%	0.5%
Dev set error:	30%	1%
	high bias	low bias
	high variance	low variance



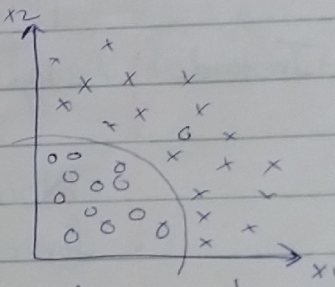
underfitting



overfitting



high bias, high variance



low bias, low variance

High bias? (training data problem) → - Bigger NN
- Train longer
- NN architecture

High Variance (Dev set problem) → - More data
- Regularizat^n
- NN architecture

Previously → bias-variance trade-offs

modernly → big data & big N/w → low bias, low variance → is possible

Regularization (helps over fitting)

In logistic Regression,

we tried to minimize the cost function i.e $\min_{w,b} J(w,b)$

i.e

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

λ = regularizat^n parameter

L_2 Regularizat^n. $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

for L_1 Regularization, add $\frac{\lambda}{2m}\sum_{j=1}^{n_x}\|w_j\| = \frac{\lambda}{2m}\|w\|_1$ → w will be sparse

λ will be Regularizat^n parameter

In neural Network

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

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

call it "Forbenius norm"

Instead of L2 regula^n

Now we know that to calculate dw, we need $\frac{\partial J}{\partial w}$, for in this case, it will be:

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

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

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

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

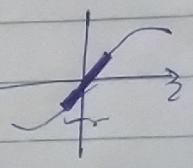
"weight decay"

because $w^{[l]}$ is multiplied by $\frac{\alpha\lambda}{m} \times 1$ and we subtract it

Why does Regularization help in less overfitting?

Intuition-1: It lessens the impact (weights) of some hidden units. So if they are hampering in the o/p it reduces. Thus regularization penalizes weights being too large.

So, in intuition, it could be seen that if $\lambda \uparrow$, then $W^{[l]} \downarrow$ which means $Z^{[l]} \downarrow$ ($Z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$)
for function ~~that~~ tanh, $g(z)$ will roughly be linear



Hence, the whole network will become linear. If such is the case, the NN will not be able to compute complex decisions
So, if regularizat^n becomes large, $W \downarrow$, activation = tanh $\rightarrow$ overfit

A debugging tool is to plot J v/s # iterations it should be 
if not, regularizat^n is needed

Dropout Regularization

Consider that each layer has 0.5 chances of keeping the nodes and eliminating the nodes. For a particular example, randomly $n^{[l]}/2$ nodes ~~b~~ will be chosen from ~~each~~ layer L. All the incoming and outgoing ~~nodes~~ weights from these nodes will be eliminated.
Thus we will be training a small network. ~~The~~ For the next ~~set of~~ example, another set of nodes will be removed.

Implementing dropout ("Inverted dropout")

layer $l=3$, keep_prob = 0.8 (80% prob of keeping neurons)

```
d3 = np.random.randn(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)        # a3 *= d3
a3 /= keep_prob   <- is used to keep the expected value of a3 as it is
```

example 50 units → 10 units shut ~~off~~ off

$Z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}$
↑ reduced by 20%

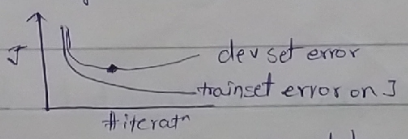
disadvantage: The size of a3 remains same during training, no need to add ~~a3~~ a3/= ?.

The results are random. You don't want random results at test time
Trying different combinat^{n}s of neurons is computationally inefficient and will also lead to similar o/p.

Why does drop-out work?
Intuition: Can't rely on any one feature, so have to spread out weights
L2-regularization shrinks weights. But it should not only rely on particular features.
Also, you may apply different keep_prob for different layers.
Generally used in computer vision where, there are a lot of features.
Should be used only when the model overfits

disadvantage: J is not well defined.
Cannot draw J v/s #iterat^n's plot.

Other Regularizat^n methods
1) Data Augmentat^n: change orientat^n of image & add data
2) Early stopping: To stop a neural n/w early when you get optimized value. that is min. J(cost).

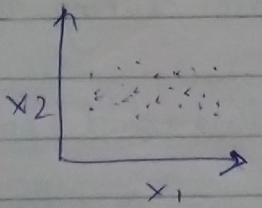


disadvantage: Taking 2 ML problems: optimize J & not overfit, to solve which adds complexity.
Advantage: With a single go over the gradient descent, we can try small, med. & large vals. of w.

Normalizat^n of Inputs

Adv: Speed up training of model

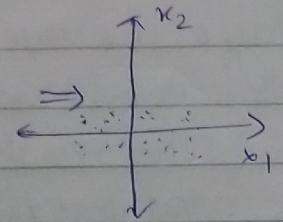
$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$



⇒ Subtract mean

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

$x := x - \mu$



Normalizing variance

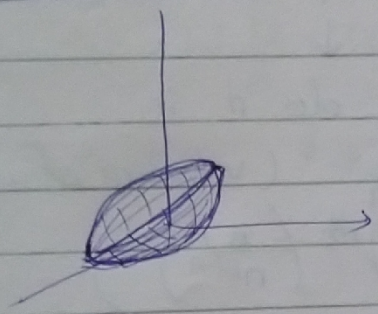
$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} **2$$

(** 2 ← elementwise)

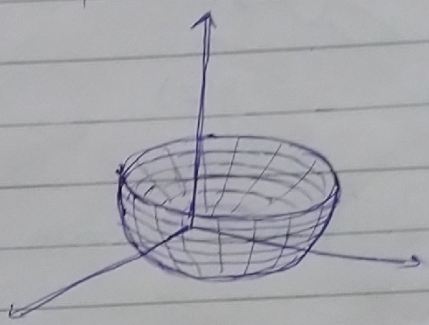
$$x /= \sigma^2$$

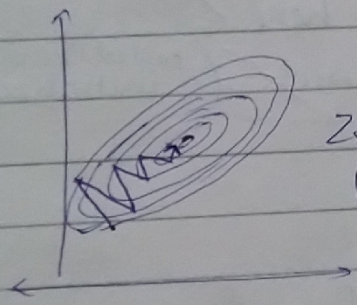
Use the same σ & σ^2 to normalize your test data, do not use a new one

Unnormalized:

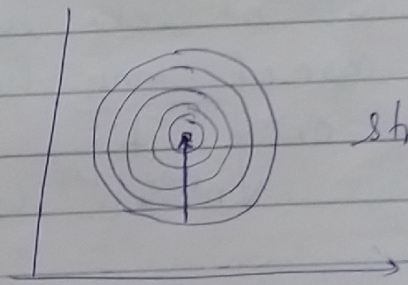


Normalized:



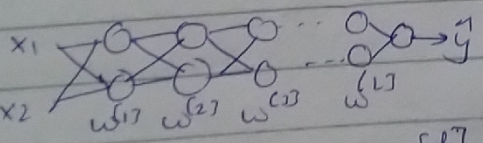


Zig-zag to zero, more time consumed



straigth to zero

Exploding/Vanising gradient



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

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

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

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

$$a^{[3]} = g(z^{[3]}) = z^{[3]}$$

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

if $w^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$

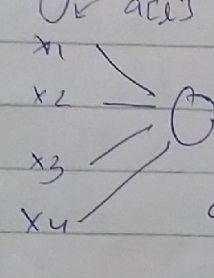
$$\hat{y} = w^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} x \rightarrow 1.5^{L-1} x$$

→ expanded if it were 0.5, it were vanishing

If the gradients are exponentially small or big, it is difficult to train based on gradient descent, as it will take tiny steps.

To solve this careful weight initialization provided as a partial solutn

Weight Initializatn for Deep Networks



$z = w_1x_1 + w_2x_2 + \dots w_nx_n$

in order to keep z in a normal range,
if $n \uparrow \Rightarrow w_i \downarrow$.
One way to do it is,
$Var(w_i) = \frac{1}{n}$ (variance of weights is set)

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

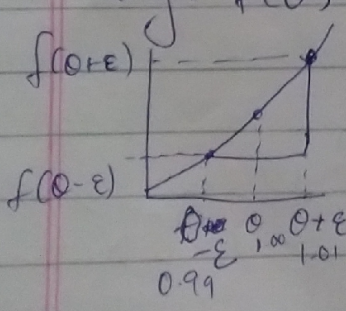
in case of relu function, it is 2 here
in case of tanh funct^n $\sqrt{\frac{1}{n^{[l-1]}}}$ is used. (called xavier initializatn)
or $\sqrt{\frac{1}{n^{[l-1]} * n^{[l]}}}$ (some autors use this)

You could also add another parameter to this variance, for hyper parameter surge. It might have a slide side effect too

Train NN's faster

Checking your derivative computatn

e.g. $f(\theta) = \theta^3$



$g(\theta) = \text{slope} = \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} = 3.0001 \approx 3$

Actual $g(\theta) = 3\theta^2 = 3$ ⟶ correct

Gradient Checking : Tool to find bugs in back propagat^{n}.

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

(Concate)

$J(w^{[1]}, b^{[1]} \dots w^{[L]}, b^{[L]}) = J(\theta) = J(\theta_1, \theta_2, \theta_3 \dots)$

similarly, $dw^{[1]}, db^{[1]} \dots dw^{[L]}, db^{[L]} \Rightarrow d\theta$

Grad check

For $d\theta^{[i]}_{approx} = \dfrac{J(\theta_1, \theta_2 \dots \theta_i + \varepsilon, \dots) - J(\theta_1, \theta_2 \dots \theta_i - \varepsilon, \dots)}{2\varepsilon}$

should be $\approx d\theta^{[i]} = \dfrac{\partial J}{\partial \theta_i}$

to check this, we take euclidean vector btwn.

$d\theta^{[i]}_{approx}$ & $d\theta^{[i]} = \dfrac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2}$

consider we take ε of order 10^{-7}

in that case, if o/p is $\approx 10^{-7}$ great!

$\approx 10^{-5}$ ok

$\approx 10^{-3}$ wrong

Tips

1. Don't used in training - only to debug
2. If algo. fails grad check, look at components to try to identify bu
3. Remember Regularizat^{n} $d\theta$ = gradient of J (add λ thing)
4. Doesn't work w/ dropout → in such cases, keep., keep-prob = 1.0 do grad check and then put ↓ = 0 anything
5. Run at random initializat^{n}; perhaps again after some training. (Not use offenly)

Optimization Algorithms : train your NN faster

$X = [\underbrace{x^{(1)}\ x^{(2)} \ldots x^{(1000)}}_{X^{\{1\}}}\ \underbrace{x^{(1001)} \ldots x^{(2000)}}_{X^{\{2\}}} \ldots\ x^{(m)}]$

(n_x, m)

Now, in case if $m = 5$ million, then it will take so long to train.

In such a case, we make baby training sets. we call them mini batches.

For example, lets make size of mini batch as 1000, we name it $X^{\{1\}}$. Here, 5000 such mini batches will be made.

Similarly we do for Y. ($Y^{\{1\}}, Y^{\{2\}}, \ldots$)

Size of $X^{\{1\}}$ is $(n_x, 1000)$