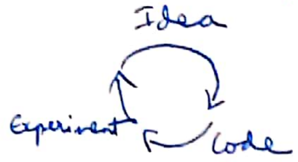
Course 2 : Improving Deep Neural Networks

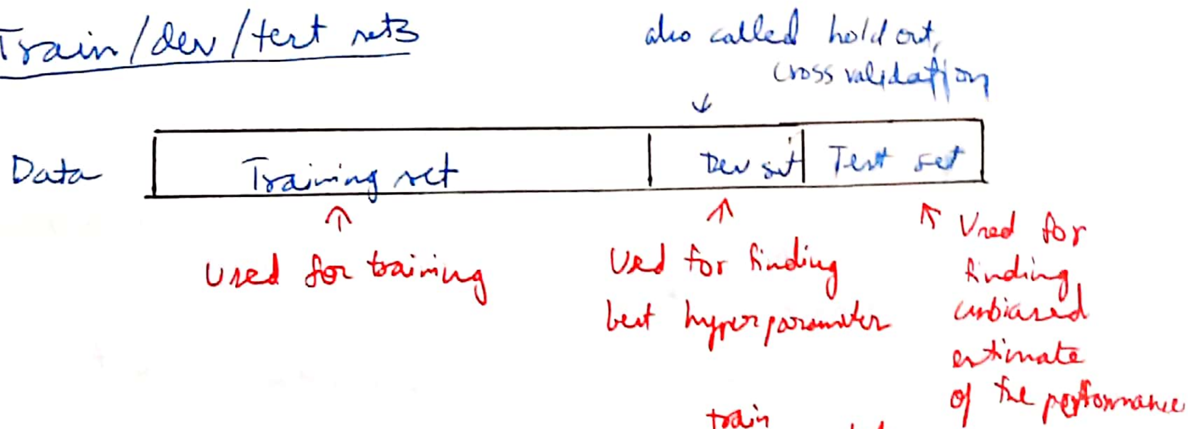
Week 1: Practical Aspects of Deep Learning

- Applied ML is a highly iterative process.



- To find the best hyperparameter (#layers, #hidden units, etc), we start off with an idea of what might be the best, write the code, experiment it and then decide a new idea that can be better....
- Intuition from one field won't work in another (eg: a person who is in NLP doing computer vision)

Train/dev/test sets



- Usually people split is as 70%/30% or 60%/20%/20%
  (train ↓ 70%, test ↓ 30%)
  and this is fine if you have less training examples (100 - 10,000)
  But if you have, lets say a million training examples then its better to divide as 98/1/1, even if its 1%, its still 10K training examples & serves the purpose

Mismatched train/test distribution

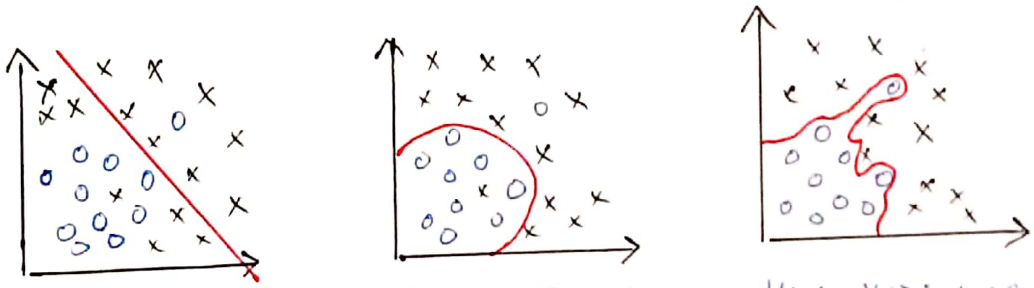
Make sure dev and test set come from same distribution

Training set:	Dev/test ~~ts~~ sets:
Cat pics from webpages	Cat pics from users using your app

Here pics maybe of better quality than this

Its fine to train from webpage pics. (like most people do with scraping), but make sure dev & test set come from same distribution

Bias / Variance



High bias — Underfitting | "Just Right" | High variance — Overfitting

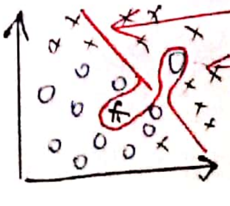
Take a classification example:

Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	High variance	High Bias	High Bias & High Variance	Low Bias & Low Variance

Usually humans have ≈0% error - Optimal/Bayes error ≈ 0%

If ~~it~~ it were 15%, the this would have low bias & variance

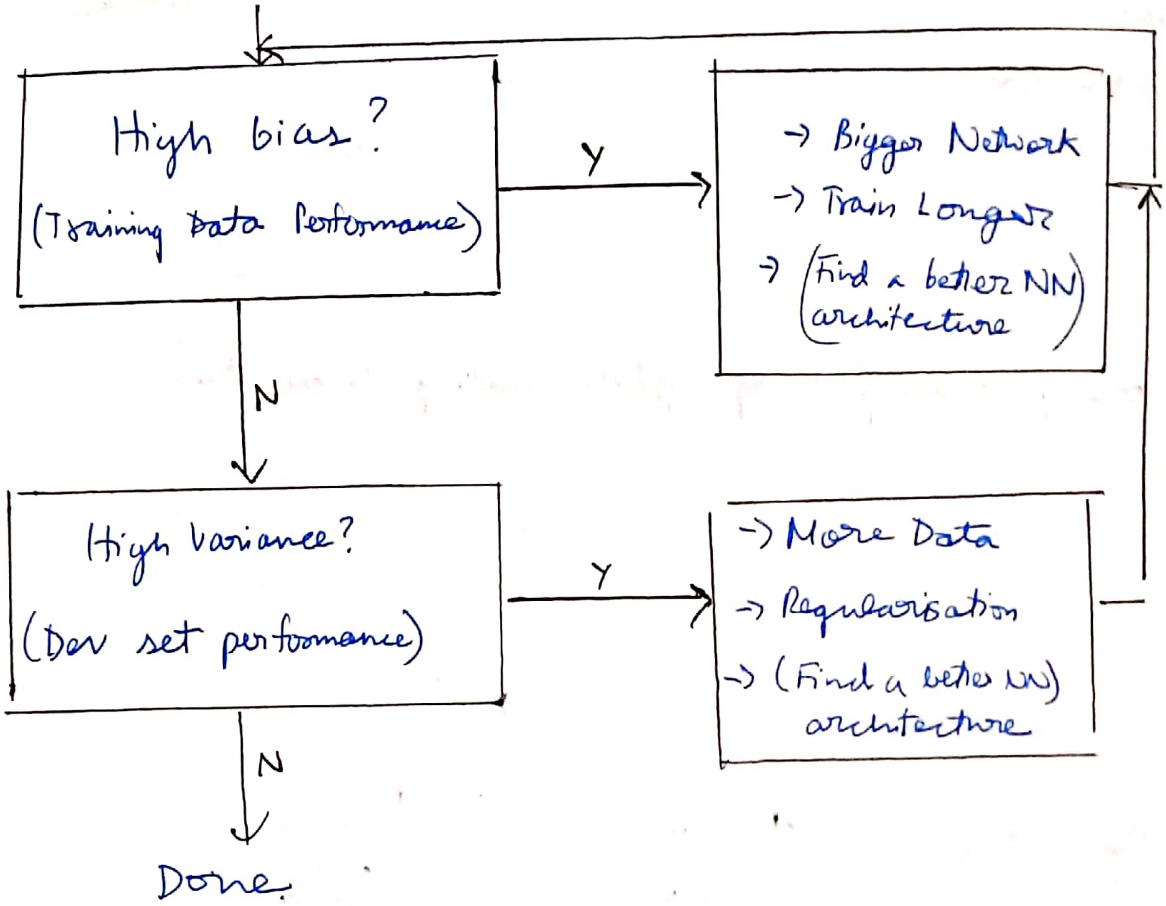
What does high bias and high variance look like?



It'll have high bias in some places & high variance in some places

This occurs a lot in high dimension models

Basic Recipe for Machine Learning



Regularization

We add the regularization term to the cost function to prevent overfitting.

regularization parameter

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

L_2 regularisation: $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ ← used more often

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

→ In logistic regression

In neural network:

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

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

If $\|w^{[l]}\|_2^2$ it is called "Frobenius norm"

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

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

This is also called weight decay

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

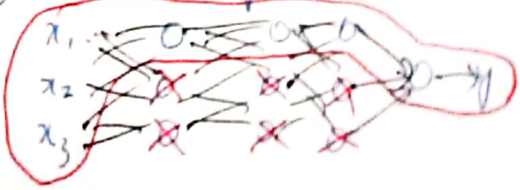
$$w^{[l]} := w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha(\text{from bp})$$

$$w^{[l]} := w^{[l]}\left(1 - \frac{\alpha\lambda}{m}\right) - \alpha(\text{from bp})$$

$w^{[l]}$ is slightly reducing (decaying)

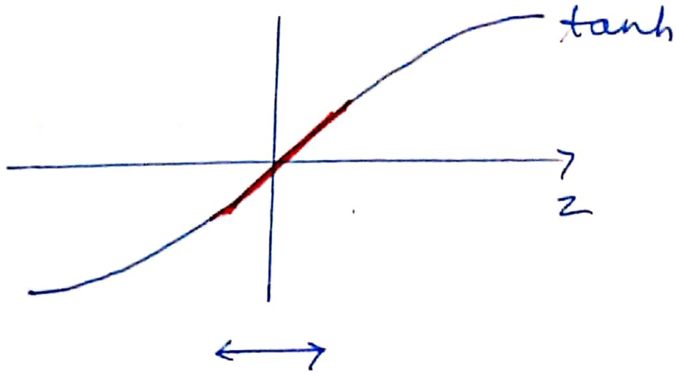
How does regularization prevent overfitting?

By penalising the $w^{[l]}$ terms, $w^{[l]} \approx 0 \leftarrow$ it starts reducing to almost 0. Therefore the neuron's start to weaken and give low outputs. This is equivalent to the NN becoming simpler network (like logistic regression)



Therefore the variance starts to reduce since the network is simpler.

Another intuition is as follows,



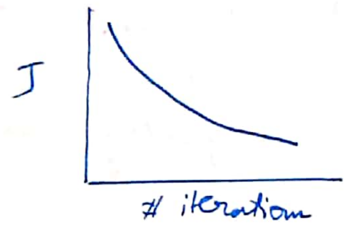
As $\lambda \uparrow$, $w^{[l]} \downarrow$ $\therefore z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]} \downarrow$

Therefore $z^{[l]}$ will be confined in the red zone where it is linear.

$\therefore$ It starts to behave more linearly (more bias)

How to debug:

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



When the regularisation term is added, J will reduce monotonically - if it doesn't, then something is wrong. If the regularisation term isn't there, the J may or may not reduce monotonically.

Dropout Regularization

Here you 'drop-out' a certain percentage of neurons at every iteration (forward + backward prop) of gradient descent.

Exa:

If ~~keep-prob~~ keep-prob = 0.8

This mean 20% of the neurons will be dropped

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

Create a matrix with 20% as 0, 80% as 1 – for this example keep-prob

a3 = np.multiply(a3, d3)   element wise

The 0 elements make 20% of a3 0 (since multiply)

a3 /= keep-prob

↑ by dividing by 0.8 $\frac{a3}{0.8}, = \frac{0.3}{8/10} = 0.3 \times \frac{10}{8}$

inverted dropout: We pump up a3 so when we do

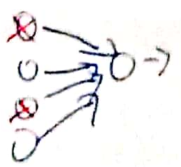
$$z^{[4]} = W^{[4]} \cdot a^{[3]} + b^{[4]}$$

The value of $z^{[4]}$ doesn't reduce (pumping up a3 compensates the 20% reduction)

Don't use drop down at test time

Why does it work?

- At every iteration, the NN is being reduced in size so only 80% of the neurons are active (in the above example)
- A neuron can't rely on any one feature, so it has to spread out weights



← Here any feature can get removed, so it will equally be dependant on all the features

- You can set different keep-prob for every layer. So this way you can choose which layer to penalise more

- Set keep-prob for input layer to 1.0 (Don't drop any inputs - not ideal) However this is done in computer vision (set to 0.9) since there are so many inputs (pixel values).

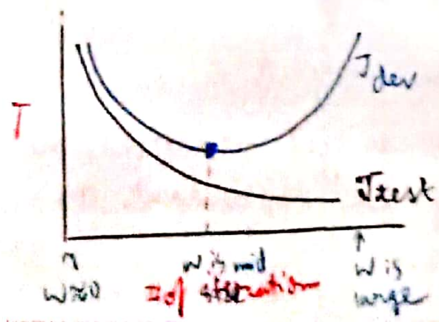
- Disadvantag:
  Harder to debug. You can't plot J with # iterations and expect a monotonic reduction 
  Instead while testing, set all the layers' keep-prob to 1.0 (Don't drop any neurons).

<u>Other regularisation methods</u>

- Data augmentation - you can flip or distort or crop existing images to get more data. This is done if you have less data and can't get more. But it isn't as good as getting new images.

- Early stopping:

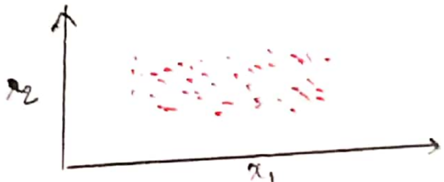
  We stop gradient descent early so it doesn't overfit the data



Disadvantage of Early stopping:

- Orthogonalisation - splitting the tasks
  - Optimise cost function
    - Gradient descent ...
  - Prevent overfitting
    - Regularisation ...

Usually we first focus on optimising cost function and then ~~pay~~ use tools to prevent overfitting, however in early stopping we are combining these 2 tasks. We don't find the optimal cost function one one hand and don't prevent overfitting properly on the other. Rather its better to seperate these 2 tasks and use appropriate ~~tasks~~ tools for that task.



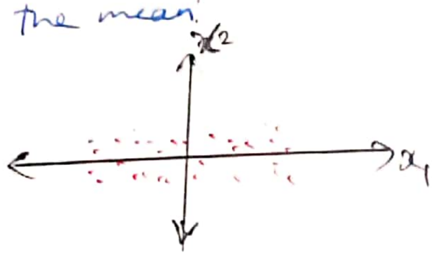
Normalizing inputs

This is used to make the data more symmetrical

→ Subtract out / zero out the mean

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

$$x := x - \mu$$



→ Normalize the variance

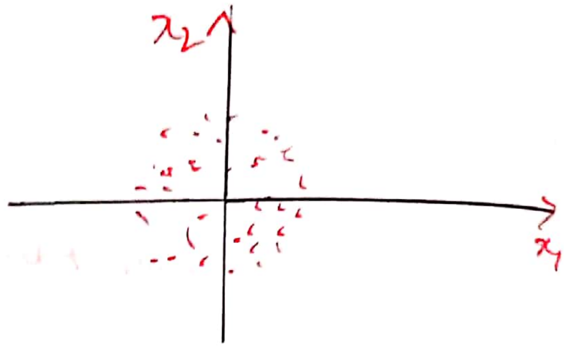
$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} ** 2$$

$$x := \frac{x}{\sigma}$$

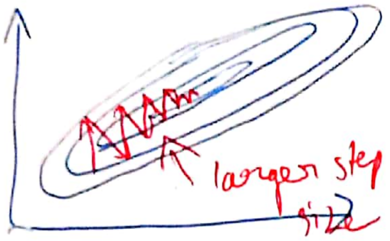
Use the same μ & σ^2 to normalise test set

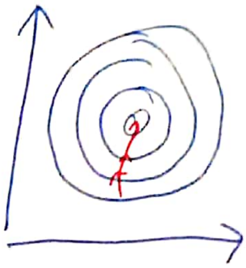
∴ Combining both,

$$x := \frac{x-\mu}{\sigma}$$



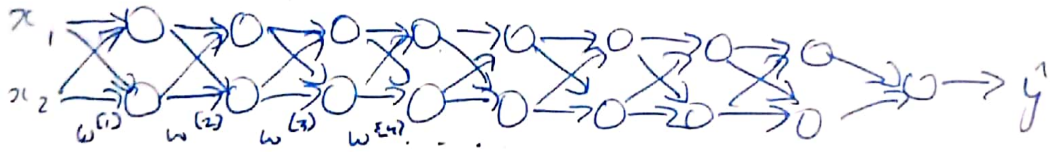
By doing this, we can use a smaller gradient descent step





Vanishing/Exploding gradients

- One problem with deep NN is that the data explodes (increases exponentially) or vanishes (decreases exponentially).



Let $g(z) = z$ - linear function and $b^{[l]} = 0$

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

a^2

a^3

. . .

① Suppose $w^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$

$\therefore \hat{y} = 1.5^L x$ ← Here if L is more, then data will explode

② Suppose $w^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$

$\hat{y} = 0.5^L x$ ← Here if L is more, then data will vanish

Identity matrix

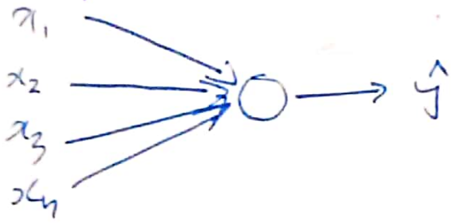
$\therefore\ w^{[l]} > I$: Data exploder

$w^{[l]} < I$: Data vanisher

There is a partial solution for this:

<u>Weight initialization</u>

Take a single neuron for example,



let $b=0$ for this example

$$z = w_1x_1 + w_2x_2 + \dots + w_nx_n + \cancel{b}$$

Now we can see that if we don't want z to be too big, for large $n \Rightarrow$ smaller w_i

Setting variance of w_i, $var(w_i) = \frac{1}{n}$ or $\frac{2}{n}$ for ReLU

solves the problem

we do this by,

$\frac{2}{n^{[l-1]}}$ for ReLU

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

Other variants,

for tanh $= \sqrt{\frac{1}{n^{[l-1]}}}$ ← we use 1

Xavier initialization

other, $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

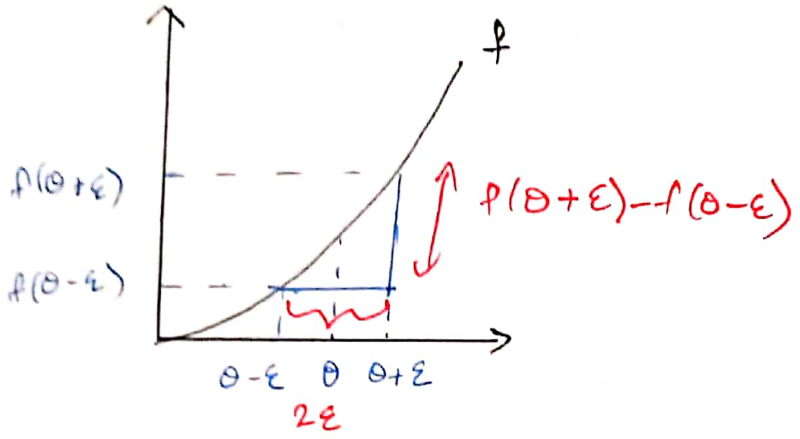
But for ReLU use this

He initialization $\rightarrow \sqrt{\frac{2}{n^{[l-1]}}}$

recommended

Numerical Approximation of gradients

Let $f(\theta) = \theta^3$



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

Example, for $\theta = 1$ and $\varepsilon = 0.01$

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

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

Gradient Checking

1] Take $W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ

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

2] Take $dW^{[1]}, db^{[1]}, \ldots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$

Now we use this to check if $d\theta$ is the gradient of $J(\theta)$

3) Grad Check:

for each i:

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

$$\approx d\theta[i]$$

Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$

If we get of the order, $\boxed{10^{-7} \text{ great!!}}$

10^{-5}

for $\varepsilon = 10^{-7}$ $\rightarrow$

10^{-3} ← bug is there

Gradient Checking implementation notes

1. Don't use in training - only to debug
2. If algorithm fails grad check, look at components to try to identify bugs
3. Remember regularization
4. Doesn't work with dropout
5. Run at random initialization, perhaps again after some training.

Scanned with CamScanner