So multiplying by 0.01 would be reasonable to try.

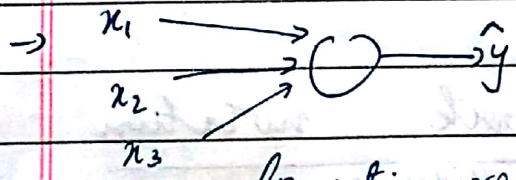
And same for w 2.

→ When we have small NN then 0.01 will work ok.

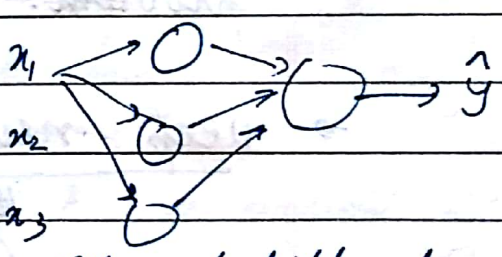
→ But when training a very deep NN, then you might need to pick up a diff constant than 0.01.

Week 4

What is a deep Neural Network

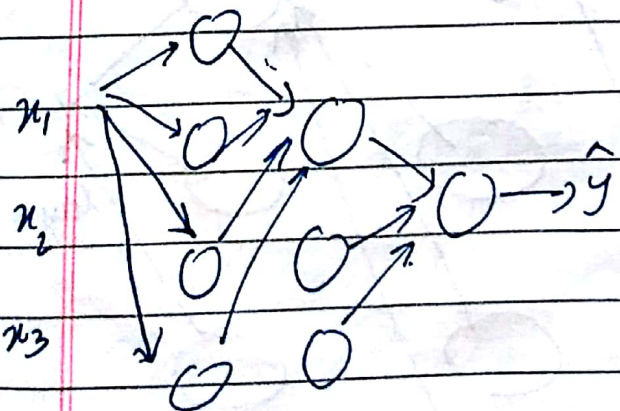
→ 

logistic regression
1 layer NN

→ 

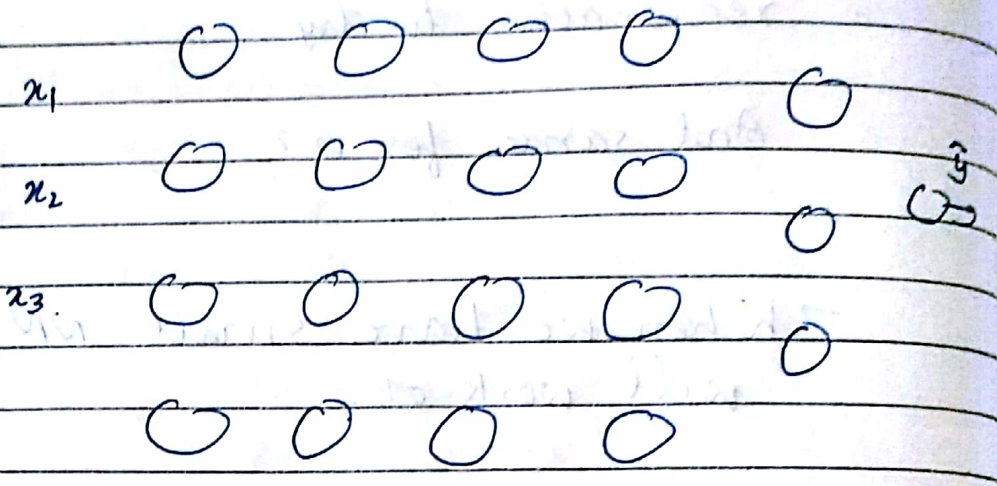
NN 1 hidden layer
So 2 layer NN.

→ NN with 2 hidden layers



Date : ..../..../.....

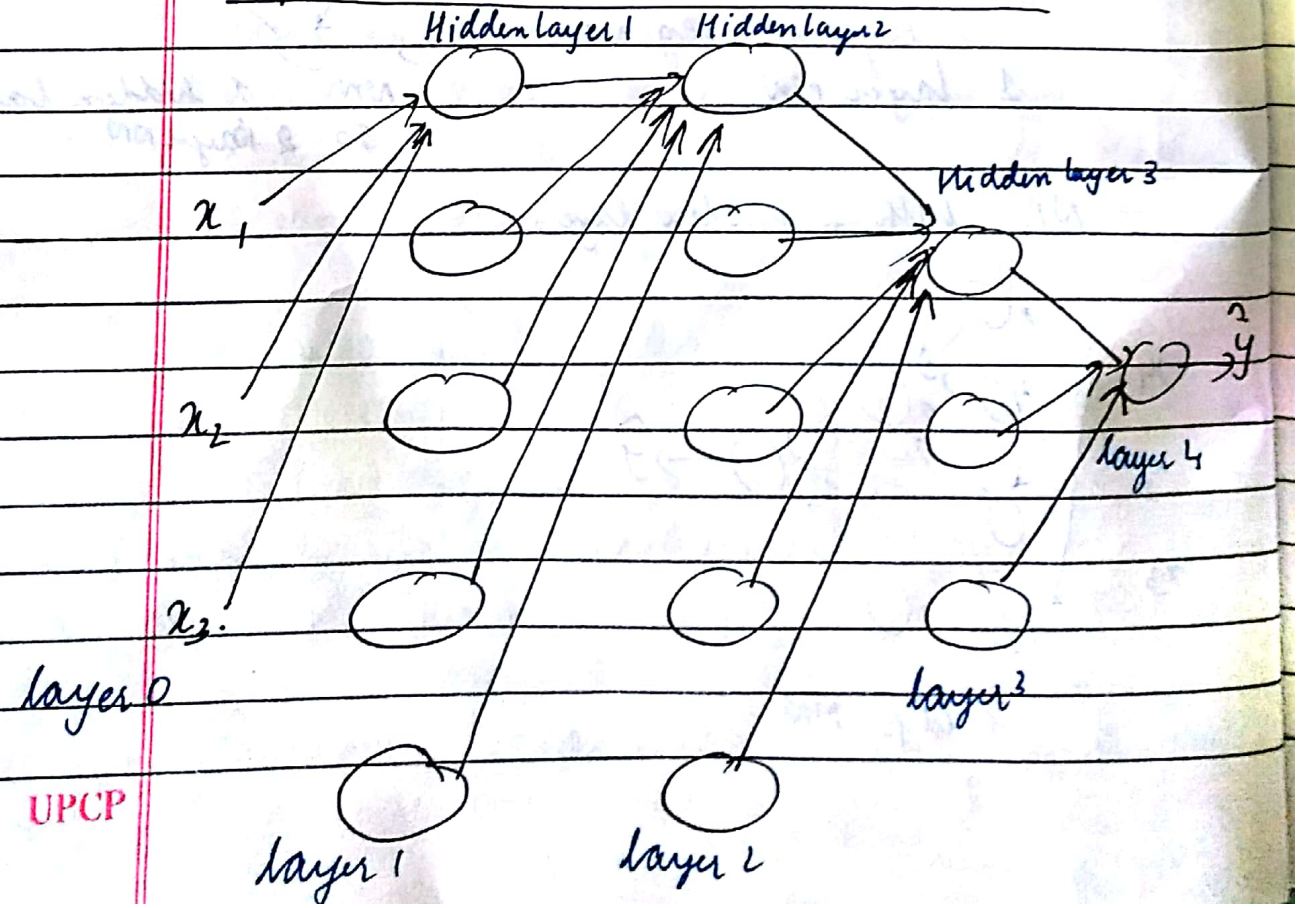
→ NN with 5 hidden layers.



* logistic regression is a very shallow model whereas as NN with 5 hidden layers is a deep model.

→ It is difficult to tell how NN with how many layers is used for a problem.

→ Deep neural network notation



UPCP

Date : ..../...../.... ..

→ 4 layer neural network.

→ It is a NN with 3 hidden layers. Hidden layer 1 has 5 hidden units. Hidden layer 2 has 5 hidden units. Hidden layer 3 has 3 hidden units.

$L = 4$ // % denote the no of layers in the ntw.

* $n^{[l]}$ = // % denote no of units in layer l.

$n^{[1]} = 5$. Hidden layer 1 has 5 units
$n^{[2]} = 5$ " " 2 "
$n^{[3]} = 3$ " " 3 " 3 "
$n^{[4]} = n^{[L]} = 1$

$n^{[0]} = n_x = 3$ // Input layer.

* $a^{[l]}$ = // % denote the activations in layer l.

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

$W^{[L]}$ = weights for computing values in $z^{[L]}$
$b^{[L]}$ = biases used — —

* The I/P features are called x. x is the activation of layer 0.

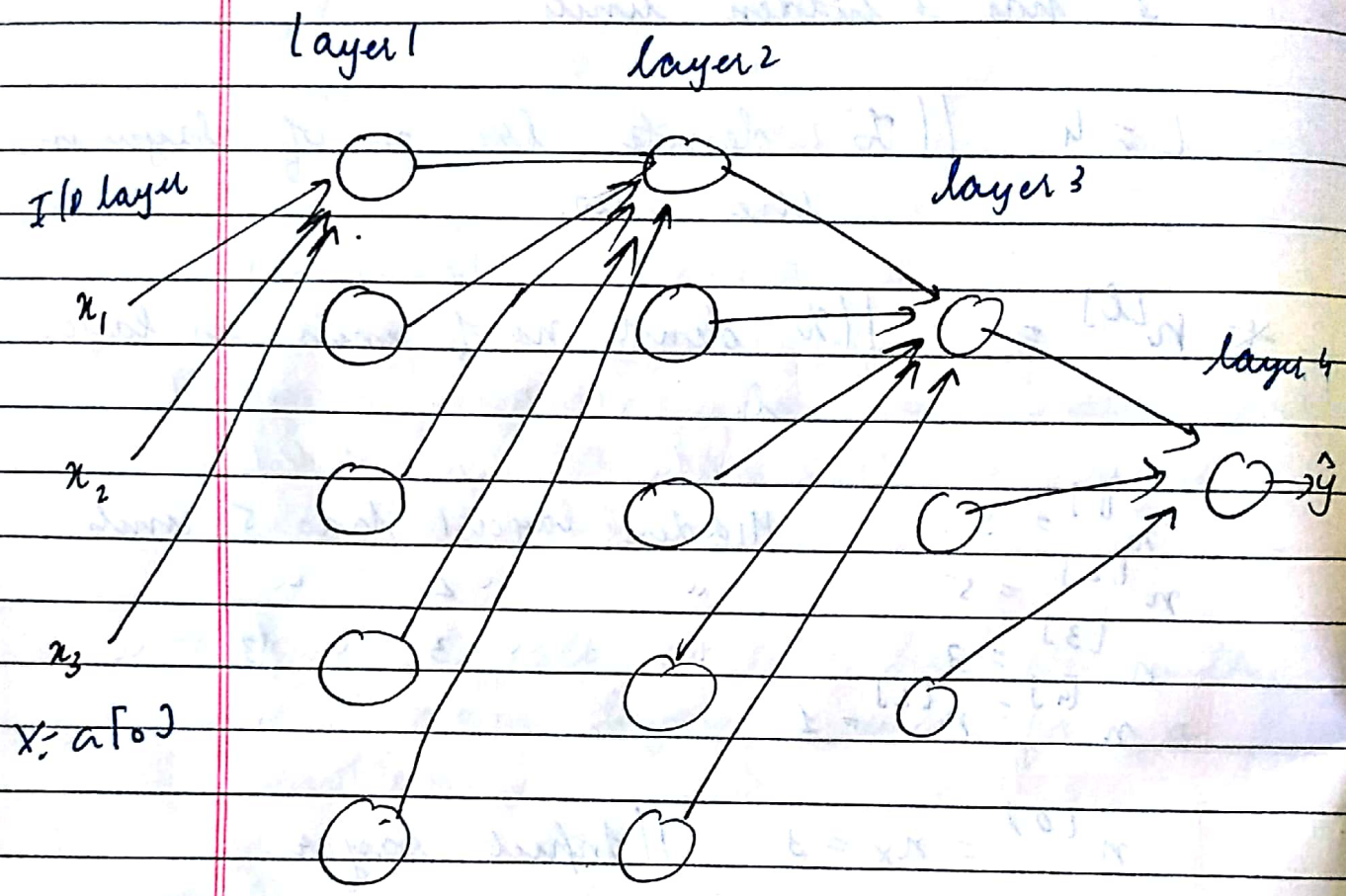
So $X = a^{[0]}$

$\hat{y} = a^{[L]}$

UPCP

Date : ..../...../.....

FORWARD PROPAGATION IN A Deep Network



Lets see forward propagation for 1 training example.

Later on will see vectorized version where we will carry out forward prop^n for entire training set at same time.

For training eg, x we compute forward propagation as follows.

For layer 1:

X: $z^{[1]} = w^{[1]} x + b^{[1]}$ or $w^{[1]} a^{[0]} + b^{[1]}$.

where $w^{[1]}$ and $b^{[1]}$ are the parameters that affect activation in layer 1.

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

↓

depends on what layer you are at

So $g^{[1]}$ represents activ^n fun^n for layer 1.

layer 2:

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

.
.
.

layer 4

$$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$$
$$a^{[4]} = g^{[4]}(z^{[4]})$$

General rule of eq^n

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

Vectorized Version → for entire training dataset.

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[1]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$
$$\hat{y} = g(Z^{[4]}) = A^{[4]}$$

Or Replace x with $a^{[0]}$.

Date : ..../..../.....

$$Z^{[2]} = \begin{bmatrix} | & | & & | \\ z^{[2](1)} & z^{[2](2)} & \cdots & \\ | & | & & | \end{bmatrix}$$

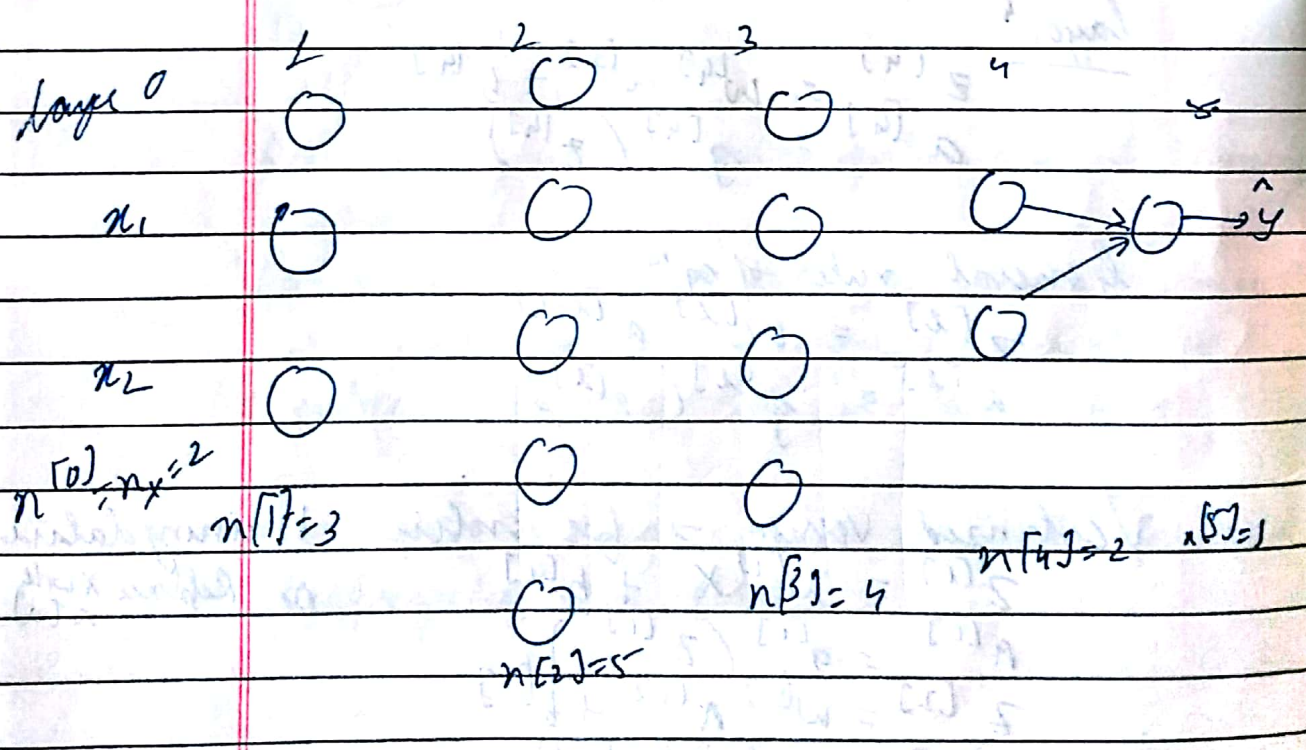
$A^{[2]}$ = Similarly.
i.e All training egs are stacked horizontally.

General vectorized form:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

Getting your matrix dimensions right.

Parameters $W^{[l]}$ and $b^{[l]}$.



$L = 5$ bcz there are 5 layers

UPCP | I I/P layer which is not counted among layers of neural ntw.

1 O/P layer.
4 hidden layers

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

(3, 1) $(n^{[1]}, n^{[0]})$ (2, 1) $(n^{[1]}, 1)$

$(n^{[1]}, 1)$ $(n^{[0]}, 1)$

dimensions of $w^{[1]}$: $(n^{[1]}, n^{[0]})$

$w^{[2]} : (5, 3) = (n^{[2]}, n^{[1]})$

Generalised dimension:

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

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

In backpropagation,
dimension of dw should be same as w
& " " db " b

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

So z & a should have same dimensions.

* In vectorized implementation,

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

$(n^{[1]}, 1)$ $(n^{[1]}, n^{[0]})$ $(n^{[0]}, 1)$ $(n^{[1]}, 1)$

$$z^{[1]} = W^{[1]} X + b^{[1]} \quad \text{// Vectorized.}$$

$(n^{[1]}, m)$ $(n^{[1]}, n^{[0]})$ $(n^{[0]}, m)$ $(n^{[1]}, m)$

m → no of training eg.

→ Broadcasting

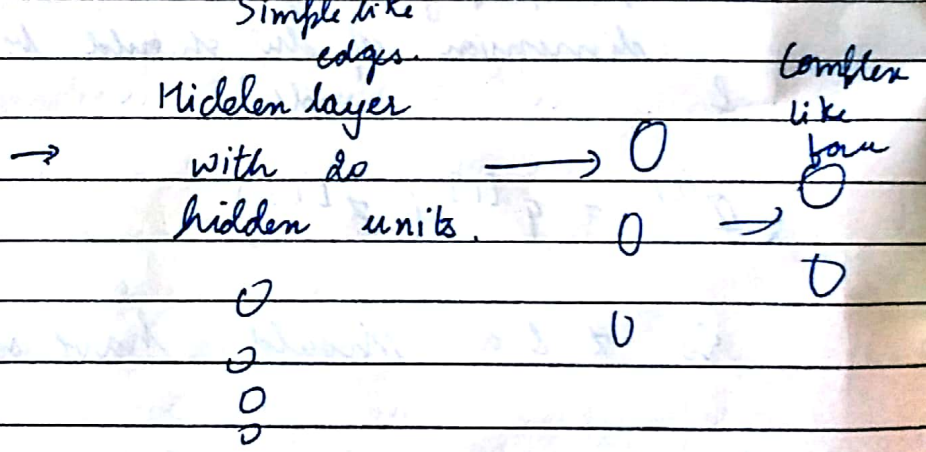
Date : ..../..../.....

dimensions of $Z^{[l]}$, $A^{[l]}$, $dZ^{[l]}$, $dA^{[l]}$ = $(n^{[l]}, m)$

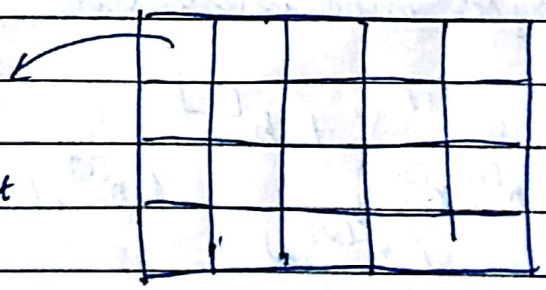
$Z^{[l]}$, $a^{[l]}$ = $(n^{[l]}, 1)$

Why deep Representations ?
Why " Ntw might work well?

1. What is deep network computing.
When we are building a system for face recognition or face detection. Here is what a deep neural network does.
I/P is the picture of face.
The first layer of Neural Network is a feature detector or an edge detector.



eg → this represents a hidden unit that is trying to visualize the orientation of edges in image.



UPCP

There are neurons to find eyes. Some neurons try to find part of the nose.

→ By putting together lots of edges it can start to detect diff parts of the faces. And finally after putting together diff parts of the faces like eyes, nose, chin, it can try to recognize or detect diff types of faces.

→ Earlier layer of neural network detect functions like edges and then composing them together in the later layers of a neural network so that it can learn more and more complex ~~structures~~ fun^ns.

* The edge detectors are looking in relatively small areas of an image.
Facial detectors look at much larger areas of image.

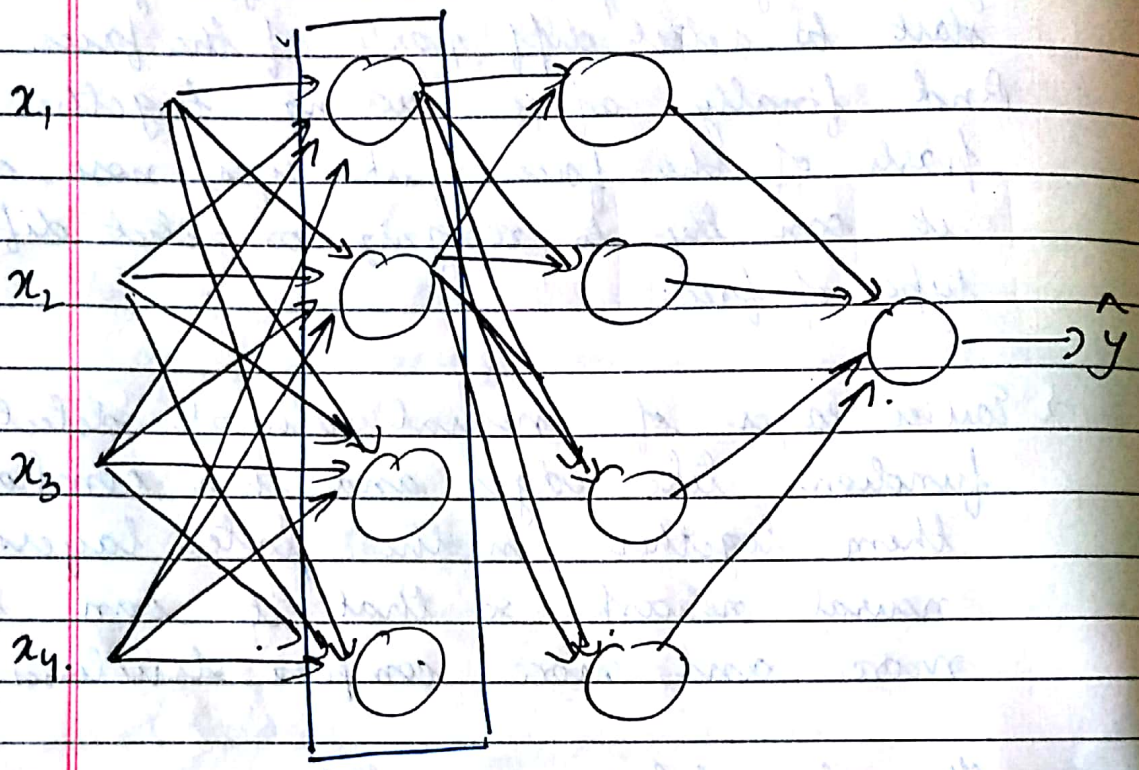
This type of simple to complex hierarchical representation or applies in other types of data than images & face recognition as well.

Circuit theory and deep learning

There are functions you can compute with a "small" L-layer deep neural network that shallower networks require exponentially more hidden units to compute.

Building blocks of deep neural network.

Forward and Backward functions:



Network of few layers is above.
Let's pick 1 layer & focus on computation on that layer for now

So for any layer, we have the parameters $w^{[l]}$, $b^{[l]}$

parameters: $w^{[l]}$, $b^{[l]}$

Forward prop:

I/P → activation $a^{[l-1]}$

O/P → " $a^{[l]}$.

for this we compute:

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

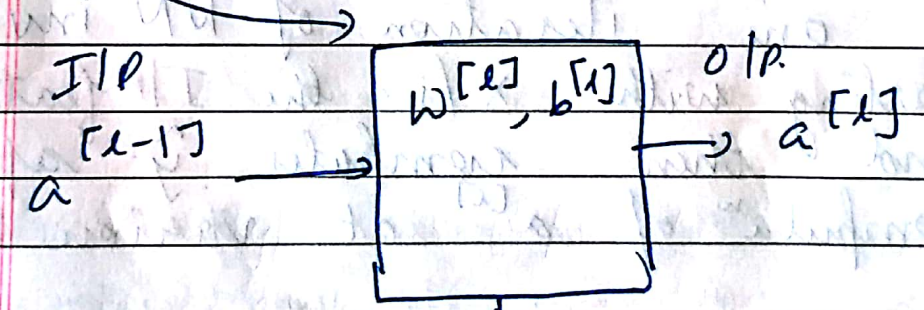
As we know $z^{[l]}$ is used later. So the value of $z^{[l]}$ is cached for later use. bcz storing the value of z. will be useful for the backward prop^n step.

Backward Propagation →

I/P $da^{[l]}$, & cache $(z^{[l]})$
O/P $da^{[l-1]}$, $dw^{[l]}$, $db^{[l]}$

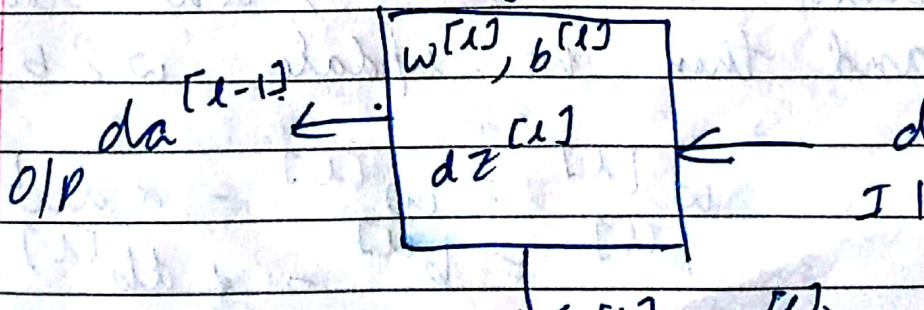
Summary

In layer l, we have forward prop^n fun^n



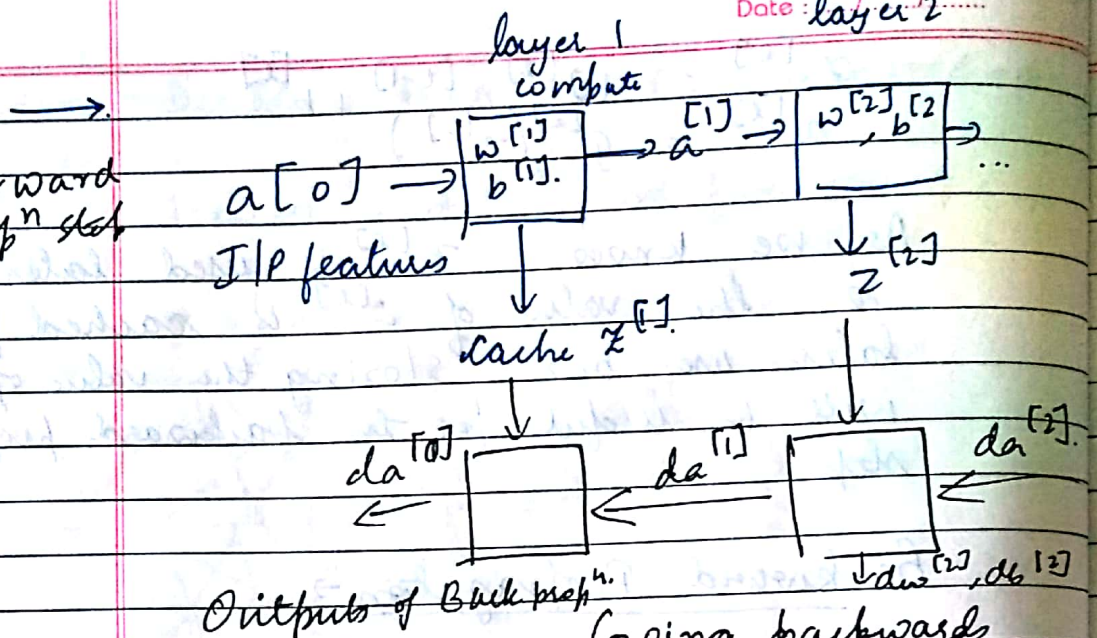
O/P Cache $z^{[l]}$.

Backward prop^n fun^n:



O/P $dw^{[l]}$, $db^{[l]}$.

Date : layer 1 ..... layer 2



Going backwards & computing gradients

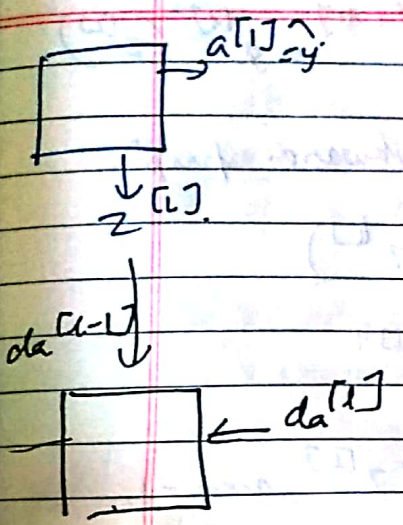
No need for derivative of I/P features in case of supervised NN

So one iteration of NN involves starting with X (i.e the I/P features) and then computes $\hat{y}$ after computa^n of $a^{[2]}$ at various layers.

And then back prop^n & then using that we have all derivative terms $dw^{[1]}$, $db^{[1]}$, $dw^{[2]}$, $db^{[2]}$... and then we update w & b.

$$w^{[1]} = w^{[1]} - \alpha\, dw^{[1]}$$
$$b^{[1]} = b^{[1]} - \alpha\, db^{[1]}$$

UPCP



Implemen
propagati

Forward p
→ Inp
→ Outp

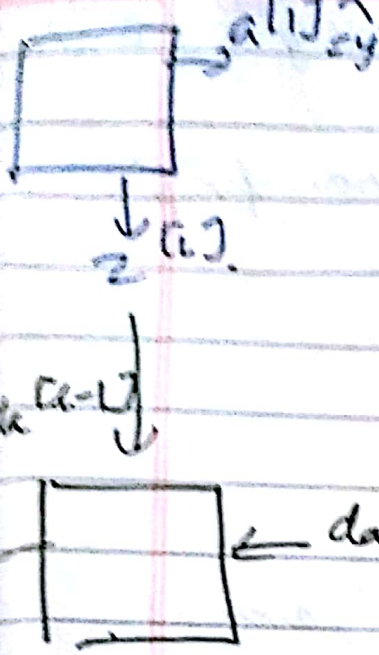
$z^{[1]}$
$a^{[1]}$

Backward
→ Inpu
→ O/P

dz

dw
db
d

UPCP



Cache is used to store the value of z for the backward functions.

It may be convenient to store w & b also in cache to get them in the backward propagation.

Implementing forward & backward propagation steps:

Forward propagation for layer l:

→ Input $a^{[l-1]}$

→ Output cache $(z^{[l]}), a^{[l]}$.

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

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

Backward propagation step for layer l.

→ Input $da^{[l]}$

→ O/p $da^{[l-1]}, dw^{[l]}, db^{[l]}$

$$dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$$

$$dw^{[l]} = dz^{[l]} * a^{[l-1]}$$

$$db^{[l]} = dz^{[l]}$$

$$da^{[l-1]} = w^{[l]T} \cdot dz^{[l]}$$

$$dz^{[l]} = W^{[l+1]T} dz^{[l+1]} * g^{[l]'}(z^{[l]})$$

Vectorized implemⁿ of backward propⁿ:

$$dz^{[l]} = dA^{[l]} * g^{[l]'}(z^{[l]})$$

$$dw^{[l]} = \frac{1}{m} dz^{[l]} A^{[l-1]T}$$

$$db^{[l]} = \frac{1}{m} np.sum(dz^{[l]}, axis=1, keepdims=True)$$

$$dA^{[l-1]} = W^{[l]T} dz^{[l]}$$

Parameters v/s hyperparameters

What are hyperparameters?

Parameters : $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]} \ldots$

Hyperpara :→ learning rate (α),

↓

determine how parameters evolve. → no of iterations of gradient descent you can carry out.

→ No of hidden layers (l)

→ ____ units

$n^{[1]}, n^{[2]} \ldots$

→ choice of activⁿ funcⁿ like relu, sigmoid.

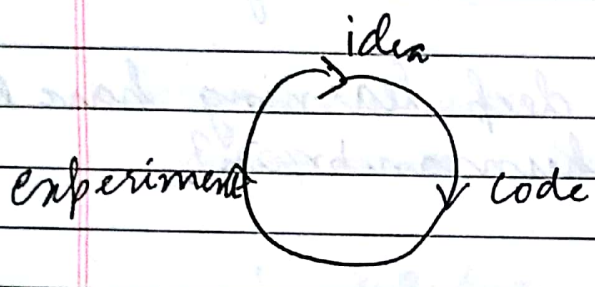
These hyperparameters control the parameters

UPCP

Date : ..../...../.... ..

Bcz they control values of W & b. Hence called hyperparameters. These hyperparameters determine the final value of W and b.

Applied deep learning is a very empirical process.

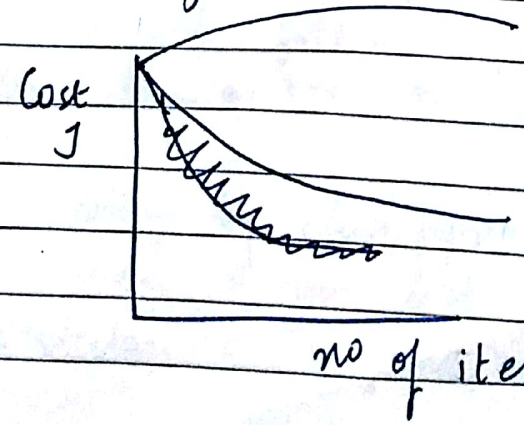


Eg → You have an idea for the best value of learning rate. You say $\alpha = 0.01$.

Then you implement it. And see how it works.

Then based on outcome, you say that i changed my mind, you increase value to 0.05.

→ If you are not sure, what is the best value of α.
Then choose 1 value of α and build cost & it goes down



On increasing α, you see cost fun^n increases & diverge.

UPCP

Date : ..../...../.....

Try other values & after trying certain values of α, it gives pretty fast learning & allows to converge to a lower cost

Empirical process bcz in deep learning we need to try a lot of things and see what exactly works.

What does deep learning have to do with human brain?

Classifying male or female given just their Body Measurements.

* Data Science is the study of data.
" Scientist is someone who solves problem by studying data.

Dependency used in this gender classification program is scikit-learn.
↓
a machine learning package with pre built models for us to use.

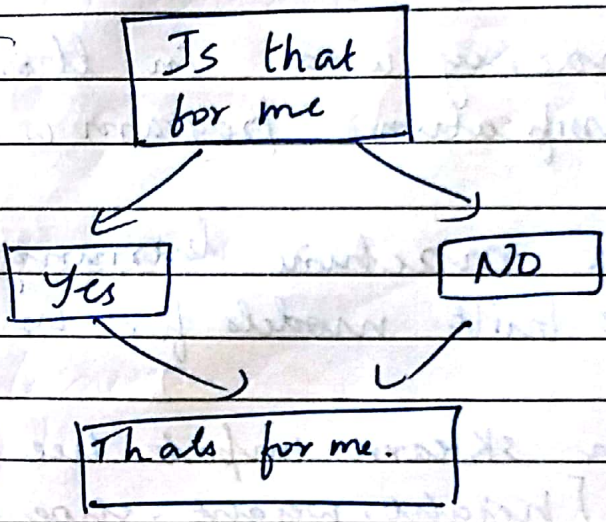
```
from sklearn import tree
# [height, weight, shoe size]
X = [ [181, 80, 44], [177, 70, 43],
      [160, 60, 38], [154, 54, 37] . . . .
    ]

Y = [ 'male', 'female', ' . . . ]
clf = tree.DecisionTreeClassifier()  // stores decision tree classifier.
clf = clf.fit(X,Y)  // fit method trains the decision tree on our data set.

// Testing by classifying gender of someone given a new list
prediction = clf.predict([[190, 70, 43]])
```

Date : ...../...../.....

Tree → It will help build a machine learning model called a decision tree.

↓

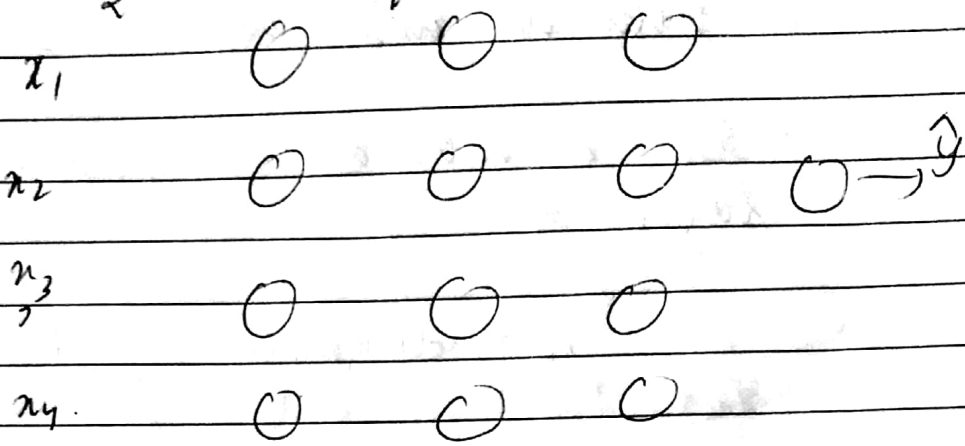
It is like a flowchart that stores data.



Date : ..../..../....

Dropout

→ Regularizⁿ Technique



overfitting

→ dropout → Go through network and say some probability eliminating a node in your network.

eg At each node we set prob of 0.5 of keeping a node & 0.5 prob of eliminating it.

So after coin tosses, we decide to eliminate those nodes & all the incoming & outgoing synapses from them. This results in smaller, diminished network

→ Train 1 eg on this much diminished network

So for each training eg you train using 1 of these new reduced egs.

→ Here for each eg training on a smaller neural network.

Date : ..../..../.....

Implementing dropout (Inverted dropout

Illustrate with layer $l = 3$

→ Set a vector $d3$ = dropout vector for layer 3

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

same size as $a3$

Let keep prob = 0.8

// It Generates random vector. & compares with threshold value.

$a3$ = activations computed.

$a3 = np.multiply(a3, d3)$

$a3 = a3 * d3$.

In this some nodes gives 0 activn.

→ $a3 /=$ keep prob // This is used for scaling

If 50 neurons in hidden layer.

50×1 → dimension of $A3$.

UPCP

Date : ..../..../....

→ At test time, No drop out

$a^{[0]} = X.$

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

$a^{[1]} = g^{[0]}(z^{[1]})$

No need to eliminate hidden units.
If you implement dropout in test time, it just adds noise in your predictions

During training neural networks is doing well but during testing it never gets correct O/P.

→ eg students who mug up for exams score rubbish in a real exam.

→ Dropout → Creating a N/w where each neuron has chance to be activated or disabled
It helps to create N/w with reliable units.

eg → It is more beneficial to consult a committee of doctors than a single doctor.

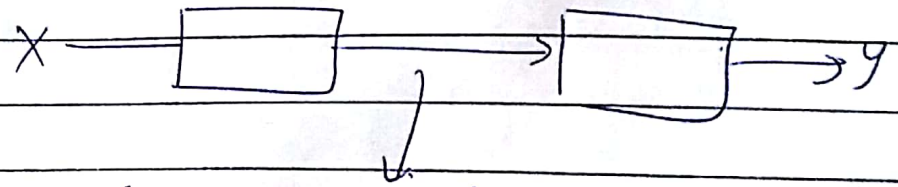
→ It gives the avg of a very larger no of possible neural networks, & we only have

UPCP

Date : ..../..../.....

we have to train / Network.

→ Dropout is slow in comparison to training / network.



Activations are values that go from / layer to other.

Some activations are somehow set to 0.

So in this way o/p does not rely on any given activation

So for the same I/P even if we don't have few hidden units, o/p remains same.

Everything remains fine at the end bcz even if few hidden layers are smashed, there are some other hidden layers that do the same task.

Dropout

→ Randomly eliminates hidden units. So for each iteration it seems a it

keras / optimization

Date ../../..

We are training on a smaller neural network.

- 1 unit can't rely on any 1 I/P feature bcz any 1 of its I/P can go away at random.

→

Course 2

Date : ../../..

Improving deep Neural Networks:

Hyperparameter tuning, Regularization & optimization.

Practical aspects of deep learning

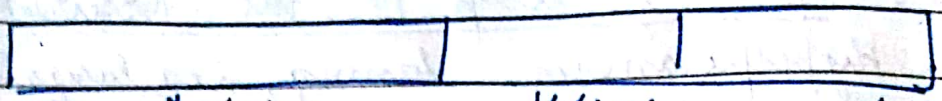
→ Making good choices in how you set up your training, development and test sets can make a difference in helping you find a good, high performance neural network.

→ Some decisions for training a neural network
- how many layers does your neural network have.
- how many hidden units each hidden layer has.
- What's the learning rate
- What are the activation functions you want to use for diff layers

→ It is difficult to choose them in your first attempt.

→ How to make progress quickly?
Progress is determined by how efficiently you can go around this cycle.
& also set up dataset in training, development and test sets.

idea → experiment → code



Workflow

→ Keep on Training algo on training set.

→ Dev set is used to check many different models and helps to find which model performs best.

→ Take the best model found after doing step 1 & 2 for long time. This best model is evaluated on test set in order to get a unbiased estimate of how well your algo is doing.

* When solving ML problem, data is divided into train/dev and test sets;
If we have relatively smaller data set, then ratio is like 60% training, 20% test, 20% dev.
When data set is much larger, it is fine to set up your dev and test dataset to be much smaller.

eg → Earlier in ML era, the data is divided in train, dev, test data in ratio of 60, 20, 20 or 80, 10, 10 or something like this. Here data is not very very large.

In this era of Big data, the data is usually very large. So if total data is 1 billion then we need only about 10000 dev data that will be enough to help us decide which algo is best.

10000 test data is enough.

So here 98% training data, 1% dev, 1% test.

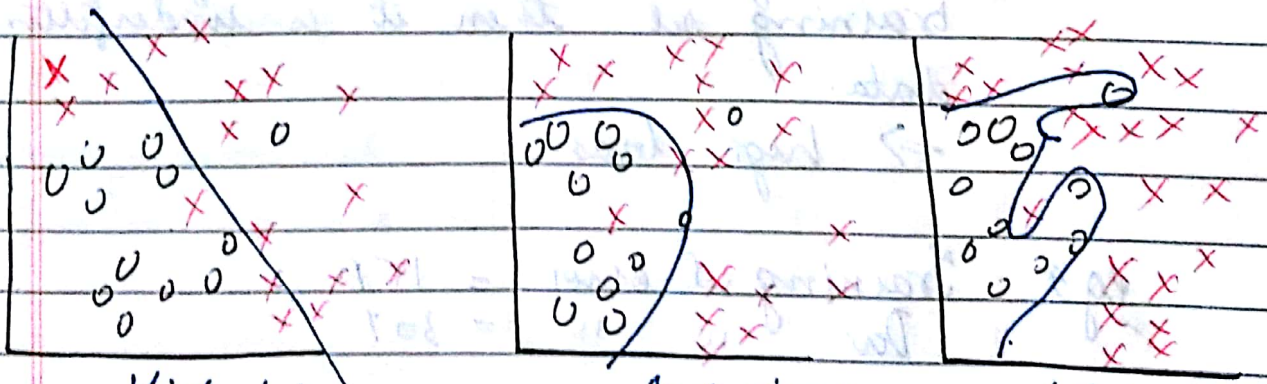
→ Mostly Models have :

Training set
→ Cat pictures from webpages. (It is in high resolution)

Dev / test set.
→ Cat pictures from users.
(It may be blurred).

Make sure that dev & test data come from the same distribution. i.e if test data is from random images from phone in casual manner by user. Then dev should also be from it.

Bias / Variance



High bias

→ Maybe logistic reg^n fit on drawing straight line. Not good fit to data. Hence high bias or underfitting

Just right

High Variance

Using big NN with many hidden units, maybe you can fit the data perfectly
→ High variance
→ Overfitting data

Date : ..../..../.....

Cat classifⁿ

y = 1 if cat
y = 0 if non-cat

To understand bias & variance, look at ⇒

eg
- Training set error = 1%. (Let's say)
- Dev set error = 11%. (" ")

We are working good on training set but since large error ~~are~~ is in dev set.
⇒ Overfitting of data.
⇒ So generalizⁿ to dev test is not done

In this eg, there is high variance.

eg. Training set error = 15%. (Let's assume)
Dev " " = 16%.

Here algo is not even working good on training set then it is underfitting data.
⇒ high bias.

eg → Training set error = 15%.
Dev " " = 30%.

high bias. & high variance.

eg. Training set error = 0.5%
Dev = 1%.
low bias, low variance

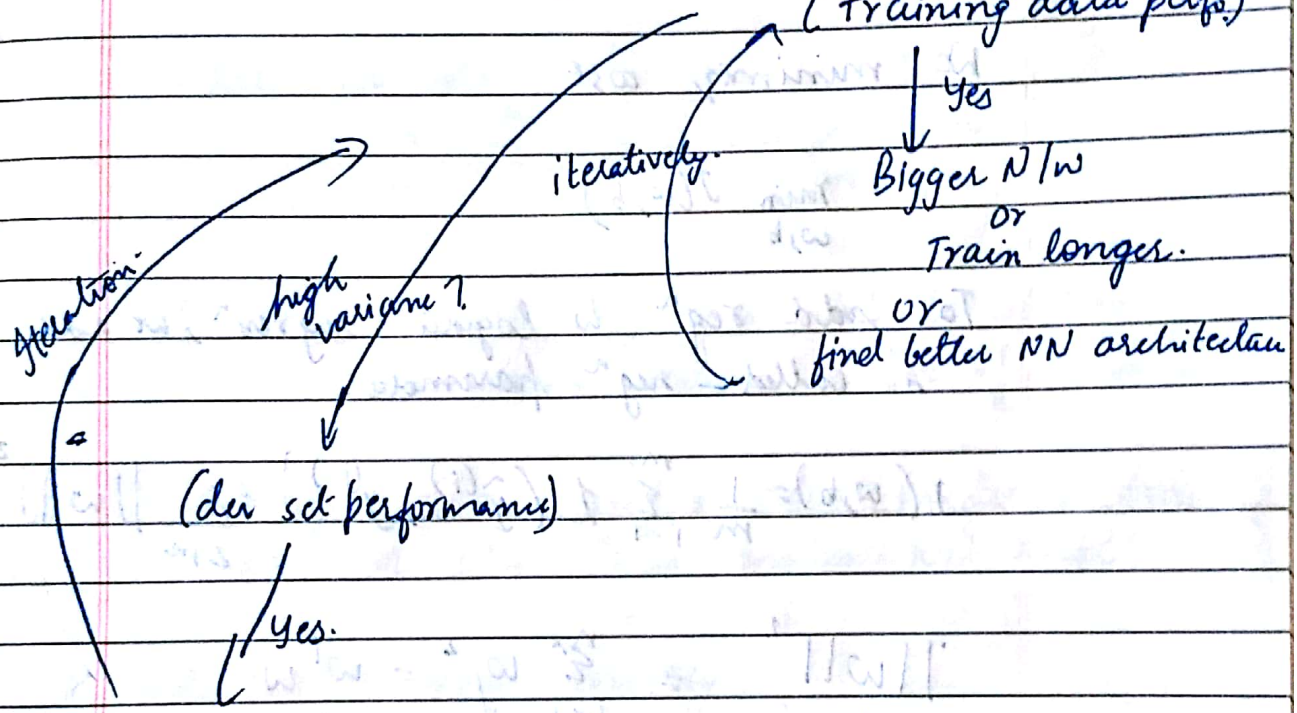
UPCP

Date : ../../....

Based on training error we can sense of bias
" " Dev " " ——— variance

Basic Recipe for ML

→ First train the model
→ check whether algo has • high bias ? (training data perfo.)



Regularizing your NN

→ If our NN is overfitting data i.e if it has high variance problem ; ⁂ Methods to avoid this are:
→ Try regularizn
→ Get more data → This is not possible always

How regularizⁿ works:

Logistic regⁿ:

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

$w \in R^{n_x}$, $b \in R$.

We minimize cost.

$$\min_{w,b} J(w,b)$$

To add regⁿ to logistic regresⁿ, we add λ called regⁿ parameter.

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

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

↓

norm w^2

This is called L2 regularization because here we are using Euclidean norm or also called L2 norm with parameter vector w.

→ Now why we regularize just a paramda w.

Why not we add $\frac{\lambda}{2m} b^2$.

↓

We can add. But since w is a high

B is a single no. So almost all parameters are in w. So b won't affect much.

L2 reg^n is the most common type of regularizn.

L1 reg^n

here we add

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

If we use L1 reg^n, w will be sparse i.e w vector will have a lot of zeros in it.

L1 reg^n may compress the model as has lots of zeros. Thus makes NN small.

* L2 reg^n is used mostly.

λ is called reg^n parameter. It is set using to dev set

λ is hyperparameter.

Neural Network

→ L2 reg^n

In NN, cost fun^n is $J(w^{[1]}, b^{[1]} \dots w^{[L]}, b^{[L]})$

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

Date : ..../..../....

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

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

called as forbenius norm of a matrix denoted by f.

↓

means sum of squares of elements of matrix.

Q. How to find grad descent

→ First dw from backprop

$$dw^{[l]} = \frac{\partial J}{\partial w^{[l]}}$$

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

After adding reg^n term to cost fun^n, $dw^{[l]}$ changes

so $$dw^{[l]} = \frac{\partial J}{\partial w^{[l]}} + \frac{\lambda}{m} w^{[l]}$$

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

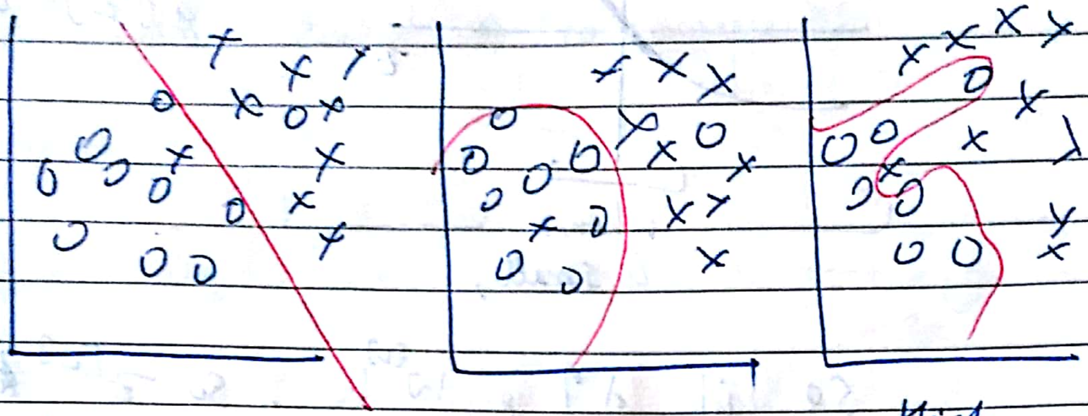
For this reason, L2 regularization is sometimes called weight decay.

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

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

UPCP

How regularisation prevent overfitting



high bias → Underfitting

Just right

High variance → overfitting

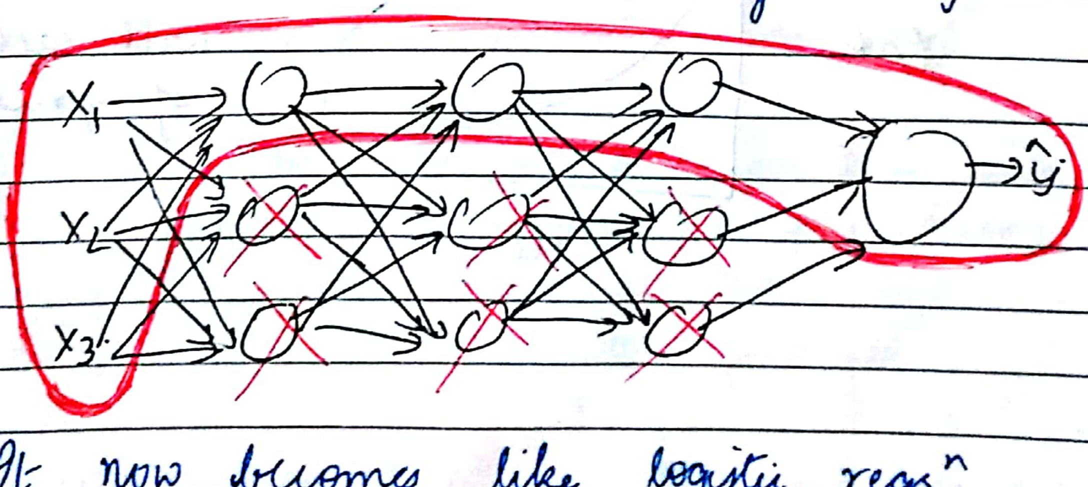
$$J(w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$$

(with L2 regularization)

Now if λ is too large, $w^{[l]}$ for many hidden units become very small & they become close to zero.

$$w^{[l]} \approx 0.$$

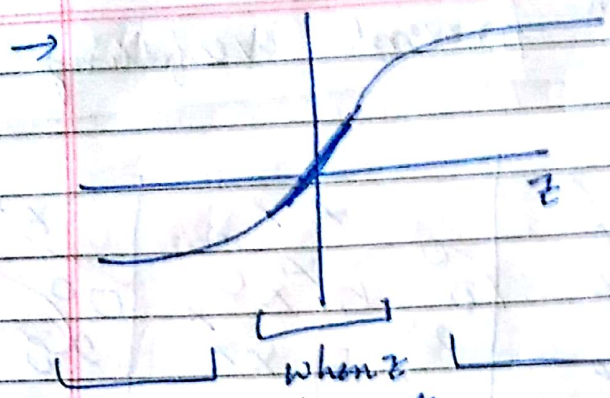
So our NN reduces to logistic regression



It now becomes like logistic regression. So for high λ, high variance case reduces to low bias case. So intermediate value of λ should be chosen.

tanh activation fun^n

$g(z) = \tanh(z)$

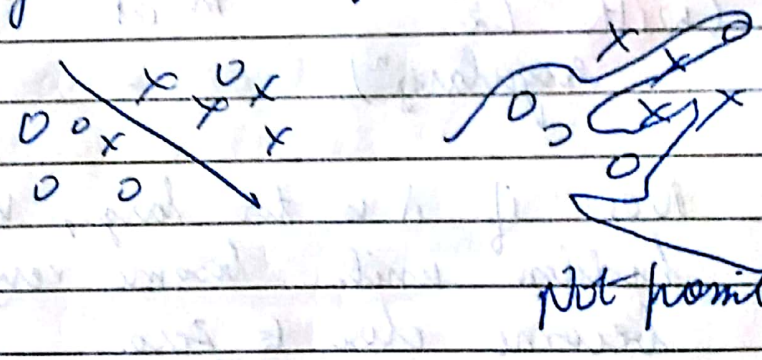


when z is small,

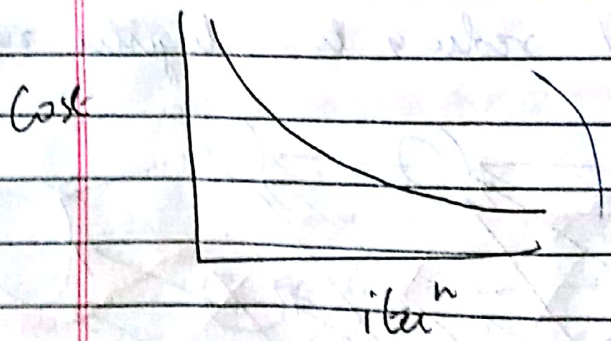
So if $\lambda \uparrow$, $W^{[l]} \downarrow$. So $z^{[l]}$ also take small values bcz

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

Hence for this activation fun^n, results will be as in linear reg^n & so no curving and overfitting can occur.



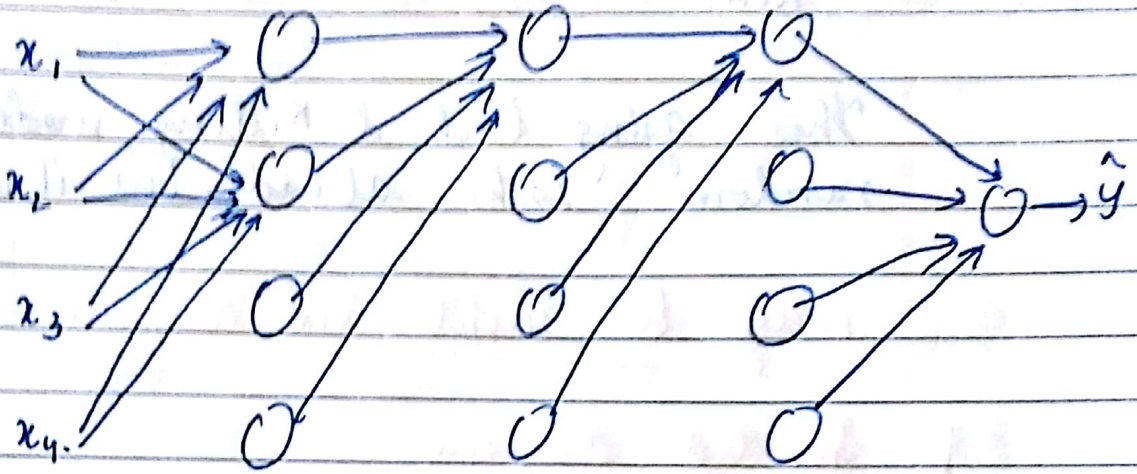
Not possible



cost decrease with each iter^n of grad descent.

Dropout Regularizⁿ

→ Powerful regulⁿ technique.



When we train using this NN, it leads to overfitting.

→ To remove ~~it~~ overfitting, we do dropout
With dropout, we are going to go through each of the layers of the network & set some probability of eliminating a node in NN.
eg → for each layer, for each node, we toss a coin & have 0.5 chance of keeping the node & 0.5 chance of removing it.
So after coin tosses we eliminate those nodes. Also we remove ingoing/outgoing links from the node. Then we are left with a much smaller diminished network.
Then we do back propⁿ, training this 1 eg on this diminished network.

Date : ..../..../.....

Then for another eg, toss the coins & keep a diff set of nodes & for each training eg we train using new reduced ntw.

This going back & killing nodes at random looks absurd but it works.

Ways to implement dropout

1. Inverted dropout

Let's assume we implement it with layer $l=3$.

d3 = dropout vector for layer 3.
= np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob

↓
Threshold.

eg keep_prob = 0.8 means 0.8 chance of keeping hidden unit & 0.2 chance of eliminating it.

a3 = activations of layer 3
a3 = np.multiply(a3, d3)

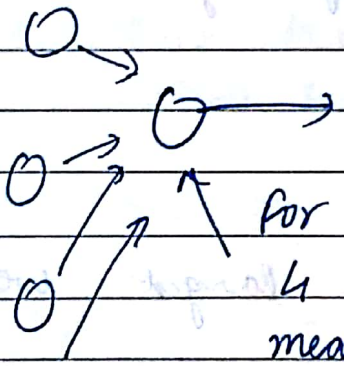
a3 /= keep_prob // This helps in scaling.

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

Making predictions at test time.

→ At test time, we do not dropout any hidden unit.

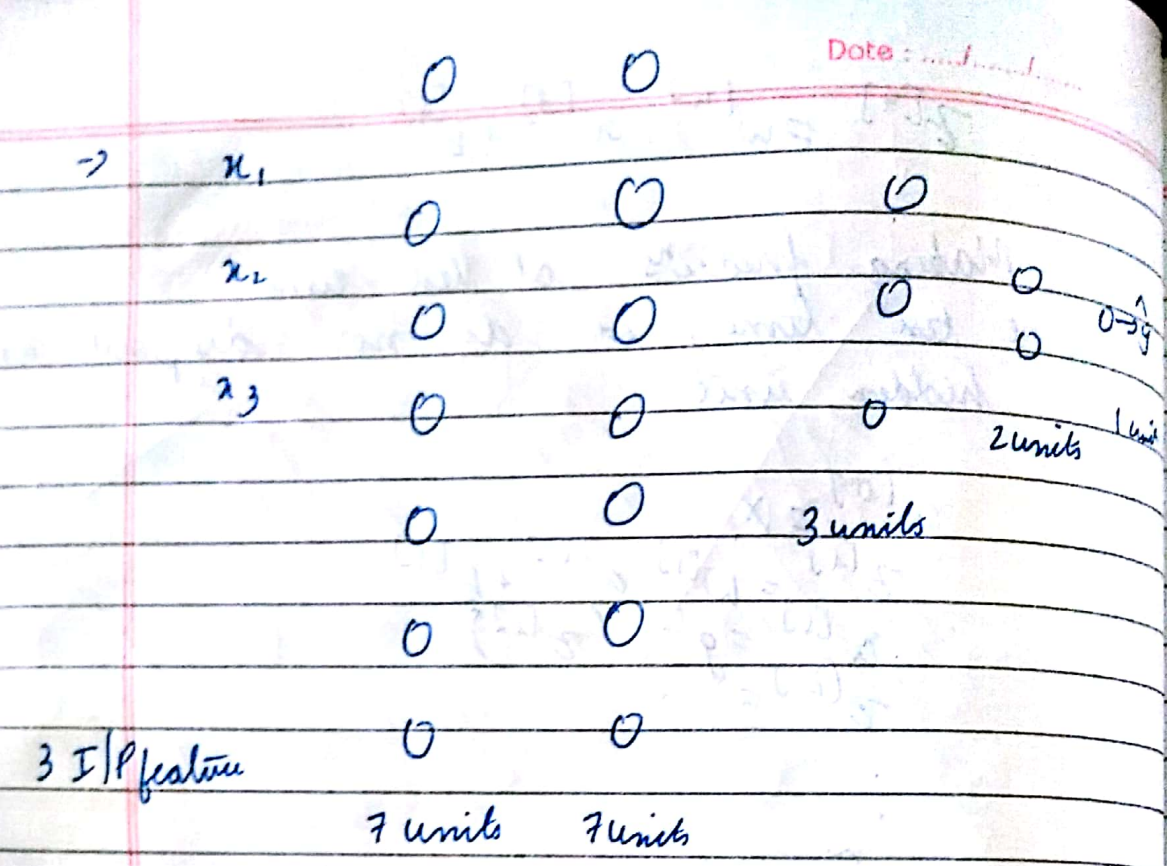
$$a^{[0]} = X$$
$$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$$
$$a^{[1]} = g^{[1]}(z^{[1]})$$
$$z^{[2]} = \ldots$$
$$\vdots$$
$$\hat{y}$$



For this hidden unit, it takes 4 I/Ps to generate some meaningful o/p.

Now with dropout, the I/Ps can get randomly eliminated. Everytime diff units may get eliminated.

So this unit can't rely on any 1 feature. bcz any feature can go at random.



→ Sometimes it is feasible to vary keep-prob layer by layer.

$W^{[1]} = 7 \times 3$

$W^{[2]} = 7 \times 7$

So $W^{[2]}$ will be largest weight matrix of NN.

So to reduce overfitting of this matrix $W^{[2]}$, we have to keep-prob that is relatively low. say 0.5.

Layers where you worry less of overfitting you can keep higher keep-probs. eg 0.7

For layers you don't care of overfitting, you can keep keep-prob of 1 which means we are keeping every unit.

UPCP