COURSE 2: PRACTICAL ASPECTS OF DEEP LEARNING

WEEK 1

TRAIN/DEV/TESTS

Applied ML is a highly iterative process

we start with ~approx hyperparameters then do

idea → code → experiment → idea

to improve them

Total
Data is usually divided into 3 parts

Train	Dev	Test

Dev ↓ cross-check

In early stages when we had less data
~ some tens thousands — we used 70 / 30 %
~~tes~~ train test

But now when we have millions of units we reduce the % of test, dev set

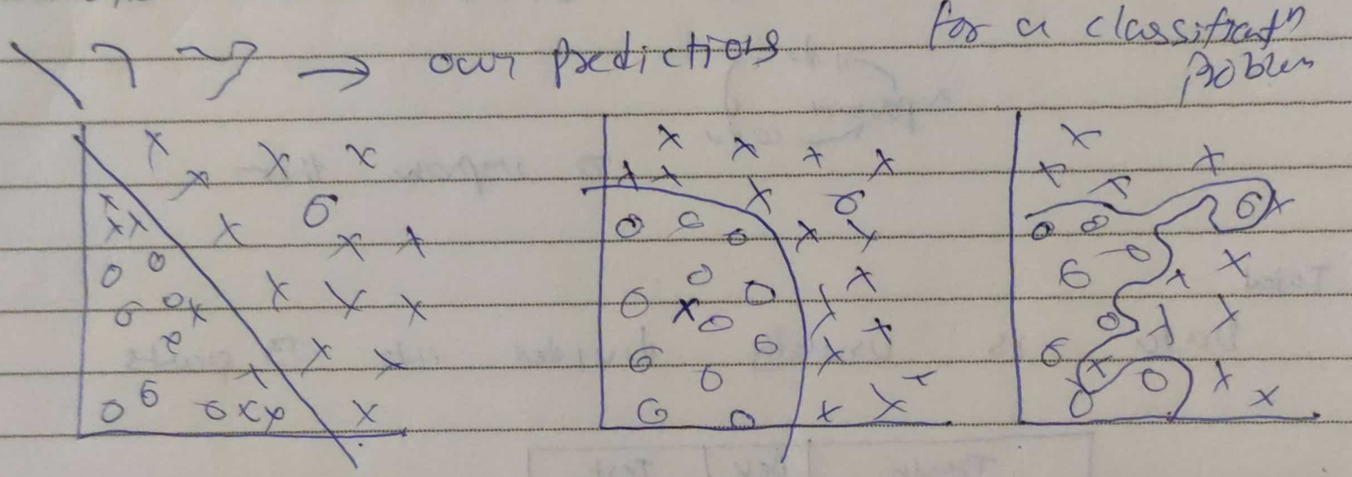
dev, Test set — used to verify which algo does better prediction
NN with diff hyper params ↓ algo

Since we have millions of datasets even a small % of them can be used to verify which algo is best

→ Test/dev and Train data should come from same distribution !! No mismatch should take place

→ It is ok if you don't have a Test set
→ you evaluate them on dev set

BIAS AND VARIANCE



Highly biased (under fitting) | Just right | High variance (over fitting)

To understand better.

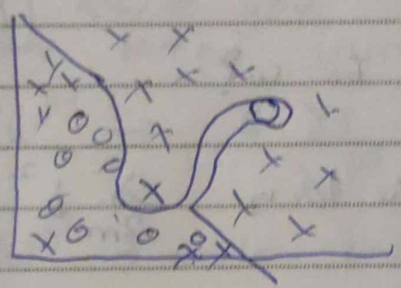
let the
Train set error : 1%
dev set error : 11% } → it means you've over fitted the model
↓
high variance

let the
Train set error : 15%
dev set error : 16% } → it means you've underfitted the model
↓
High bias

let the
train set error : 15% } → high bias and
dev set error : 30% } high variance
// bad model

let the
train set error : 0.5% } → low bias and low
dev set error : 1% } variance.

eg: of high bias and high variance



BASIC RECIPE FOR ML

1st ask
you have high bias ? $\xrightarrow{yes}$ • Try bigger network
↓ depends on train set
more iterations on Gradient descent ← train longer
use better optimization algo
// train look to a better suited NN architecture
↓ no.
then ask
high variance ? $\xrightarrow{yes}$ Get more data.
↓ depends on dev test
Regularization
Search for other better NN architecture
↓ no.

Earlier there was a tradeoff → if you try to decrease bias variance went up and vv
But now, set of things to do to increase bias/variance are diff so no trade off

REGULARIZATION

↳ use it your model is overfitting → High variance

lets understand this using Logistic regression

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

$w \in \mathbb{R}^n$

$b \in \mathbb{R}$

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

called L2 norm

we add this term to regularize w

$\boxed{\text{L2 Regularization}}$

why regularizing only for w?

Actually we can add the term $\frac{\lambda}{2m} b^2$ but it won't matter ~~that~~ much.

you can also use L1 regularization

here you add $\frac{\lambda}{m}\sum_{i=1}^{n}|w| = \frac{\lambda}{2m}\|w\|_1$

↳ if you use this then $\underbrace{w \text{ will be}}$ sparse,

there will be lots of zeros in w.

L2 regularization is used very often.

λ = regularization parameter

↓

you've to tune it

L2 regularization in Neural networks

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

$$\underbrace{\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2}_{\text{"Frobenius norm"}}$$ w is a $n^{[l]} \times n^{(l-1)}$ matrix

then how do you ~~calculate~~ apply Gradient Descent here
we did it using backprop which gave dw.

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

$dw^{[l]} \rightarrow$ whatever backprop gave $+ \frac{\lambda}{m} w^{[l]}$

correct dw when using regularization

then ~~that~~ w update becomes

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

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

$\therefore$ L2 regularization is also called "weight decay"

b/c in add^n to subtracting what we were prvsly doing we are also subtracting some ~~wt~~ multiple of w $\left[\frac{w\alpha\lambda}{m}\right]$

Why does Regularization prevent overfitting?

with large → graphical answer
→ basically we reduce down w → reduces z
↓ ↗ ↓
we penalize J for * when w is large — makes sure our sigmoid/tanh result ~~lines~~ lies around zero //

// also when you're doing regularization make sure you use the updated value for J

Dropout Regularization.

we randomly remove some nodes from layers
then we remove all the weights and incoming/outcoming edges → basically remove those node's existence
↓
→ makes our networks somewhat ~~random~~ small

But we removed nodes at random right? then why does it work?
It just does

How do we implement this?

Iterate each layer, let's say keep prob = 0.8

$d^{[3]} = np.random.rand(a^{[3]}.shape[0], a^{[3]}.shape[1]) < keep.prob$

mask ↓

we'll keep 80% of nodes.

then our a3 becomes

a3 = np.multiply (a3, d3) $\rightarrow$ $a3^{*} = d3$

a3 /= keep-prob

// dont implement dropout directly during test time

But again why does dropout (which is random) work?

Understanding dropout.

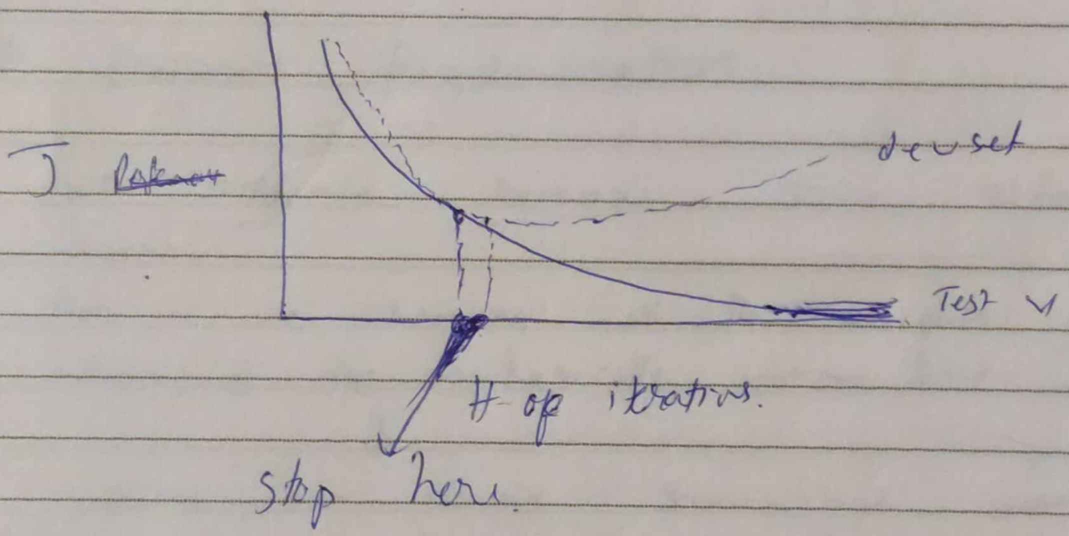
Other Regularization Techniques

→ Bigger Network (always helps)
↓

#1 data augmentation { get more data, → if you don't have access to more data generate more data using the data you already have

#2 Early Stopping (orthogonalization)

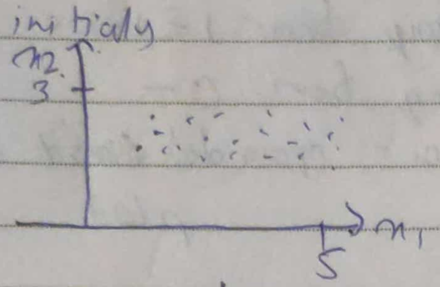
plot both test and dev set simultaneously

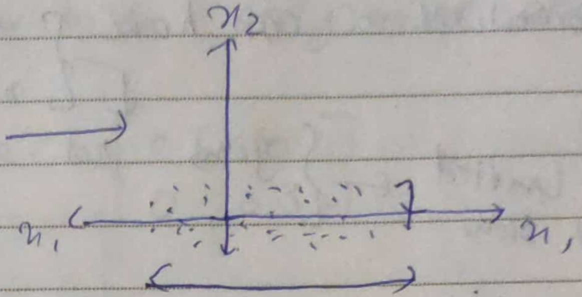


// you can just ~~use~~ use L2 regularization instead of Early stopping
↓
computationally somewhat expensive b/c you've ~~to try~~ different values of λ

Normalizing inputs
↳ helps in speeding up training

say $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$





subtract mean:

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

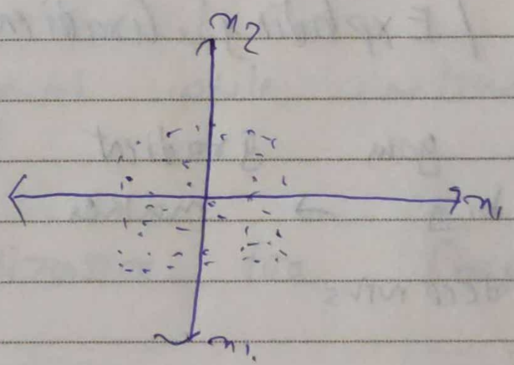
$$x := x - \mu$$

// x_1 variance is larger than x_2 //

normalize variance

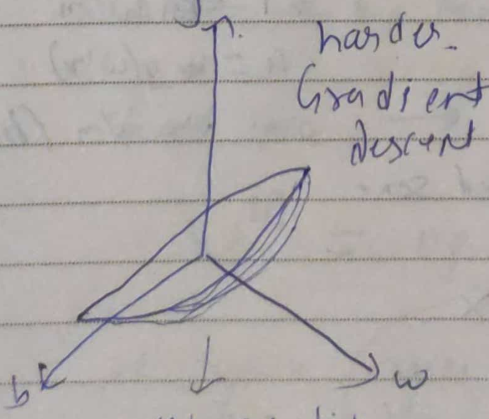
$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} x^{(i)2}$$

$$x /= 6$$

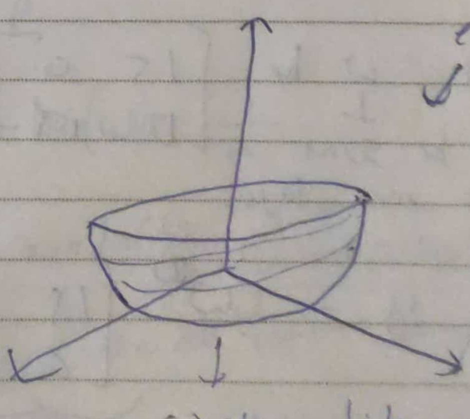


if you use μ and σ to normalize train set, use same μ and σ to normalize test set.

Normalization also helps in getting a good cost function



ellipse like
unnormalized



circle like
normalized

unnormalized can have x_1 range from 1 - 1000
x_2 range from 0 - 1

makes Gradient descent slow ← gives not such a good cost f^n plot

Normalization makes the range of all inputs similar

Vanishing / Exploding Gradients

Sometimes your gradient becomes vv small or vv big → makes training harder

specially in deep NNs

consider a deep NN with $\underset{\text{big}}{L}$ layers

let $b = 0$ for now and say we're using a linear activation f^n.

then $y = w^{[L]} w^{[L-1]} w^{[L-2]} \ldots w^{[3]} w^{[2]} \underbrace{w^{[1]} x}$

$z^{1} = w^{1} x$
$a = g(w^{1} x)$
$= w^{1} x$ (linear)

let w be $\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$ and so on

let it be same in each layer

then $y = w^{[L]} \underbrace{\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1}}_{\text{huge number}} x$

cz L is huge

this makes y v big and gradient also big

Similarly if $w = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$

y will become v small and hence gradient will also become v small

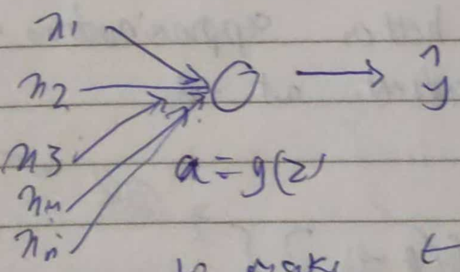
→ takeaway : dont take weight matrix of large or small no.

↓

be careful while initializing matrix randomly.

weight initialization for Deep Networks

Single neuron example



$z = w_1x_1 + w_2x_2 \ldots + w_nx_n$

large n → smaller should be the w_i

to make sure $z, a, \hat{y}$ don't blow up

Good idea → set $Var(w_i) = \frac{1}{n}$

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

↓

internal random generation makes it equal to

if you're using ReLU then $Var(w_i) = \frac{2}{n}$ works better

↓

* np.sqrt($\frac{2}{n}$) // gaussian random nd. generation ?!

np.random.randn(shape-l) x this gu $\sqrt{\frac{1}{n^{[l-1]}}}$

if you're using tanh → $\sqrt{\frac{1}{n^{[l-1]}}}$

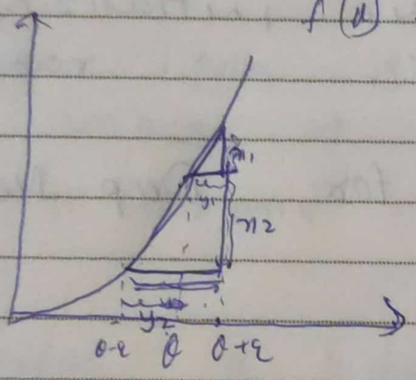
Xavier initialization

another version $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

Numerical approximations of Gradients

say you've a f^n f

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



instead of the usual $\frac{x_1}{y_1}$

$f'(\theta) \approx \frac{x_1+x_2}{y_1+y_2}$ gives a better approximation of Gradient at θ

As to formal def^n

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

and the difference b/w actual gradient and gradient calculated using $\approx$ is $O(\varepsilon^2)$

smaller ε ⟶ smaller error

if we use ... the error is $O(\varepsilon)$

relatively bad

Gradient checking

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

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

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

same dimension as θ

Is $d\theta$ same as gradient of cost function?

this is called

Gradient check (grad check)

calculate $d\theta_{approx}$

for each i

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

and you have $d\theta[i] = \frac{dJ}{\partial \theta_i}$

Now is

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

how to decide whether they're approximately equal or not?

get euclidean distance $\rightarrow d\theta_{approx} - d\theta = \| d\theta_{approx} - d\theta \|_2$

Check $\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx 10^{-7}$? yes → Great

10^{-5} ~ okayish

10^{-3} → bad → recheck your derivations

you decide (usually v small)

Gradient Checking implementation Notes

- Don't use Grad check in ~~or~~ training (is slow)
  (use only to debug)

- If algo fails grad check, look at components of dθapprox and dθ to identify bug
  - check which values of higher diff figure out which dθ and hence which dw and db are causing it.

- Remember about regularization terms

- Gradcheck doesn't work with dropout.
  ↓
  cz random.

  (you can fix which nodes to drop once then do Gradcheck)

- Run at random initialization.
  ↓
  maybe your implementation of backprop, gradient is only correct when w, b are close to 0

  so start with random initialization, run gradcheck let it train for sometime then run grad check again