Page No.
Date

Course 2: Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization.

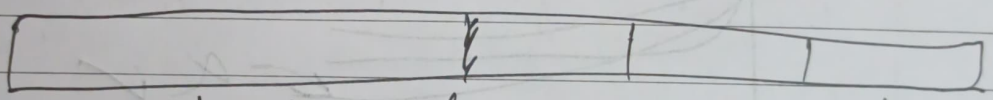
Week 1 : Setting up your ML Application

1) Train / Dev / Test sets :

Applied ML is a highly iterative process as initially you do not know #layers, #hidden units, learning rate, activation function, etc ...

DL is used in NLP, CV, Structured data

Ads, search, security, logistics



test different algos on dev set to see which one works / suits best

train ↓ test ↓

Previous era : 70 / 30 %. 60 / 20 / 20 %.

Big data era : 1,000,000 10,000 10,000

98, 1, 1 %.

Mismatched train / test distribution

eg:

Training set: Dev/Test sets:

Cat pictures from webpages ↔ Cat pictures from users using your app

* Make sure that dev and Test sets come from the same distribution

Note: Not having a test set might be okay i.e. Only train and dev set.

2) Bias and Variance

eg: Assuming Human error in classying cat/not cat ≈0%.

Train set error: 1%

Dev set error: 11% (relatively poor)

→ Overfitting the training set i.e. high variance.

Train set error: 15%

Dev set error: 16%

→ Underfitting the data (high bias).

Training set error: 15%

Dev set error: 30%.

→ Both high variance and high bias.

Training set error: 0.5%.

Dev set error: 1%.

→ Low variance and Low bias.

We classify bias and variance depending:

Optimal (Bayes) error i.e. human error

But if bayes error is higher and so is train set error then algo won't be considered as high bias

eg: Bayes error = 15%
Train set = 16%
} low bias

→ when blurry images are shown so then humans as well as machines cannot classify correctly

Note: Linear classifier has high bias
(under fits data)

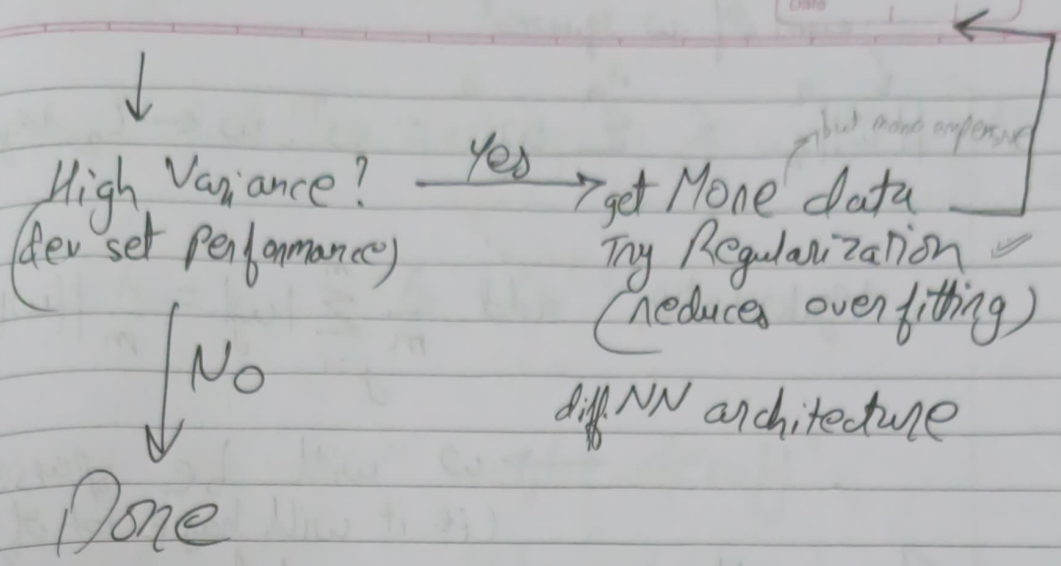
3) Basic 'Recipe' for ML

→ tells us whether the algo has bias or variance problem or both, hence improves the performance of our algorithm.

High bias? —Yes→ Try:
Bigger network
Train longer
try different NN architecture
Give access to more computation resources like CPUs, GPUs

(training data performance)

↓ No



pre-deep learning : Bias Variance tradeoff

Big data era : They both can be independently reduced without hurting each other using above methods.

Regularization :

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

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

for regularization :

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

↑ regularization parameter

omit : b just a no.

lambda → reserved keyword in python
∴ we use (lambd) → hyperparameter

Date

norm of w squared

$$\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T \cdot w \leftarrow L_2 \text{ regularization}$$

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

→ w will be sparse
(ie it will have lot of zeros in it)

This will compress our model and hence needs less memory to store the model. But this helps only a little bit hence we do not use L_1 reg.

Neural Network - $J(w^{[1]}, b^{[1]}, \dots w^{[L]}, b^{[L]})$

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

Adding regularization :

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

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

"Frobenius norm" → sum of squares of elements in the matrix

Gradient descent :

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

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

L2 reg → "weight decay"

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

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

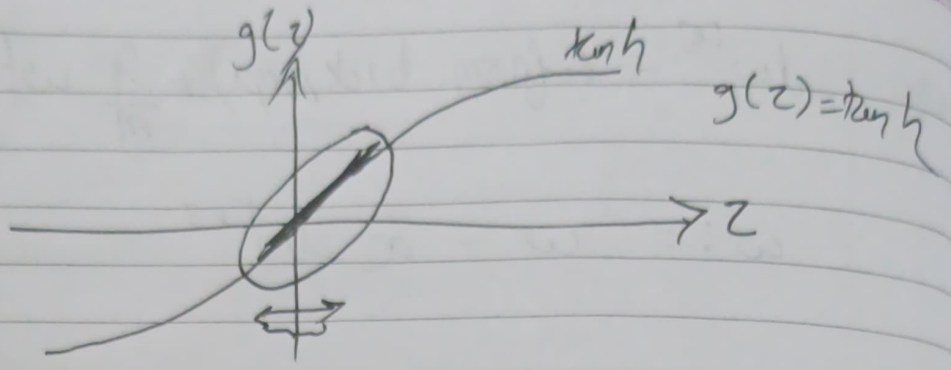
$$\left(1 - \frac{\alpha\lambda}{m}\right) w^{[l]}$$

$$< 1$$

This term effectively shrinks the weight w, on each iteration.

The regularization term $\frac{\lambda}{m} w$ increases the effective cost for large weights (by appropriately selecting large value of λ, thereby penalizing them.

This will in turn prevent overfitting as the NN will become much simpler as weights are now small hence they won't contribute much hence almost eliminating them and given the NN a structure of logistic regression

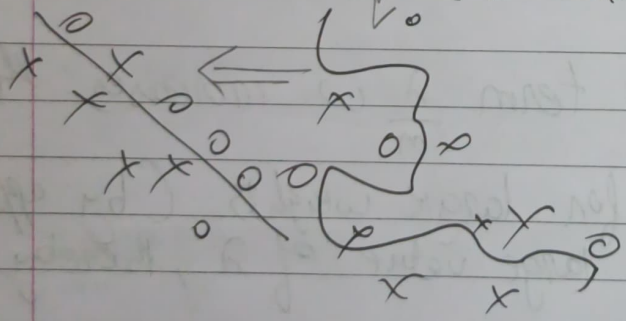


if $\lambda \uparrow$ then $w^{[l]} \downarrow$ as it is penalized due to cost f^n & grad. descent

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

$\therefore$ Every layer $\approx$ Linear

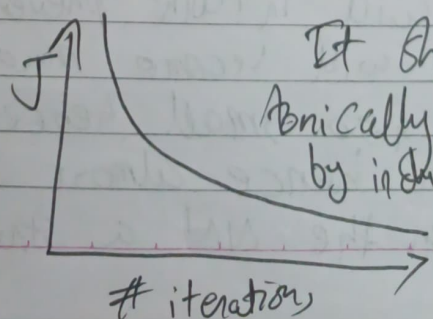
$\therefore$ The complex Deep NN just becomes a linear regression, which will only be able to compute a linear function, so it won't be able to fit complicated decisions like (non linear decision boundaries)



Hence overfitting is prevented by making complex non-linear f^{ns} to linear f^{ns}.

To debug gradient descent:

Plot cost f^n J v/s # iterations



It should monotonically decrease by iterating term $\frac{\lambda}{2m}$

↗ Helps to reduce variance of model

Dropout regularization:

Implement through "Inverted dropout" (: most common)

Illustrate with layer $l = 3$, keep-prob $= 0.8$

probability that a unit is not eliminated ↗ from l.

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

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

* `a3 /= keep-prob` → inverted dropout technique

50 units ⇝ 10 units eliminated (: prob $0.8 \times 50 \approx 10$)

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

↖ reduced by 20%. if we do not write *

Hence to compensate for the loss of hidden units and so that the output $z^{[4]}$ does not change we shall divide a3 by 0.8

i.e. in short the expected value of a3 and in turn $z^{[4]}$ remains the same as it should be ~~with cons~~ if all hidden units were present.

implemented in python as: `< 0.8`

`d3 = (d3 < keep-prob).astype(int)`

Page No.
Date

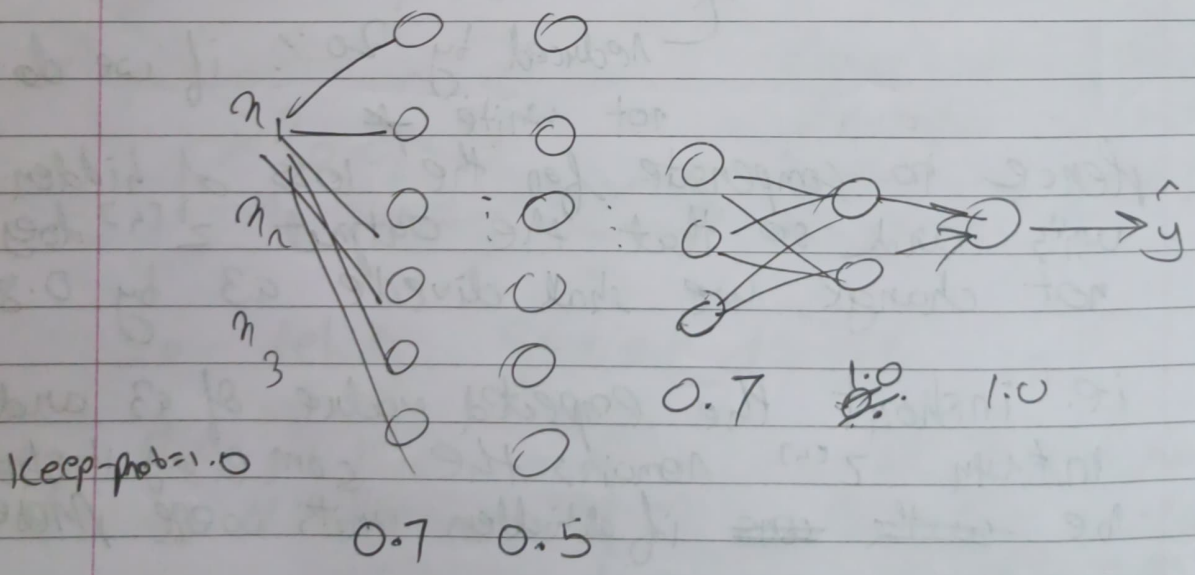
Making predictions at test time

$a^{[0]} = X$

No drop out at test time

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

We shall implement drop out during training and not testing as it would add noise to the predictions if we use it in test time.



Keep the values of keep-prob low for those layer where many hidden units are not required which

complexes the NN overall.

Dropout is used in CV as input images size is big so we do not have enough data ~~to store them~~.

1) Data Augmentation

Other Regularization tech. to reduce over fitting

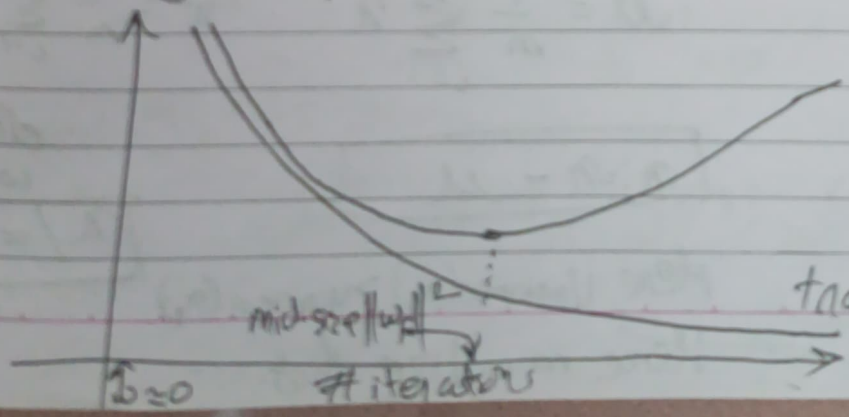
Eg: Recognize Cats / Non cats

Getting data from outside can be expensive so instead we can take an image and make a copy of it by rotating it to get a new image, so this would save us a lot. Also we can zoom into cat images and rotate them by a few degrees to make new combination of images. This also reduces overfitting.

eg: Number recognition (optical characters)

From the above example, we can implement similar methods by distorting images, rotating images*, etc.

2) Early stopping



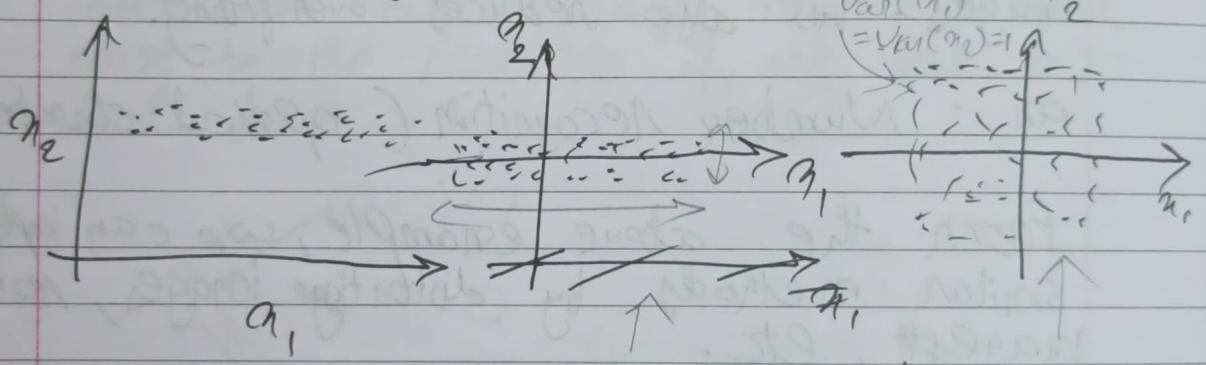
Stopping midway when dev set error is least hence to not continue the grad. descent further and extract mid-way w, b.

Downside:
By using early stopping, we are not fully using gradient descent so that is not going to fully minimize the cost function.

Alternatively we can use L_2 Reg. instead of early stopping.

Setting up your optimization problem

1) Normalizing Inputs
eg: To input features x_1 & x_2:



Subtract mean:

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

$$x := x - \mu$$

Normalize variance

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

element-wise square

$$x /= \sigma$$

Here Variance(x_1) > Variance(x_2)
Hence normalize that.

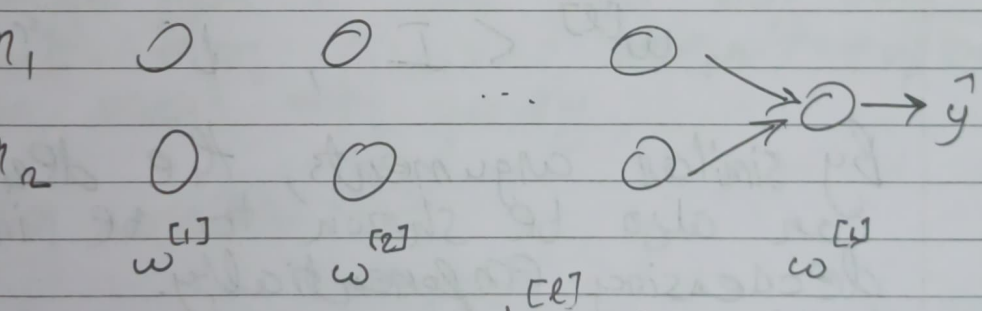
Use the same μ and σ for the test set as well, do not calculate that seperately for test set because we want the test set to go through the same transformation defined by same σ and μ calculated on training set.

Why do we Normalize ??

→ Unnormalized inputs: lead to an elongated cost function and ineffecient gradient descent path which makes it oscillate to many times to reach the minimized loss.

→ Normalized inputs: Leads to a spherical cost function and effecient gradient descent path hence the algorithm quickly leads to the minimum of cost f^n resulting in consistent step sizes in all directions

2) Vanishing / Exploding Gradients:



$g(z) = z$

$y = w^{[l]} w^{[l-1]} \dots w^{[3]} w^{[2]} w^{[1]} n$

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

$a^{[2]}$

$a^{[3]}$

$$w^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \quad \text{where, } l = 1, 2, \ldots$$

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

This is an exponential function (1.5^{l}), so as the NN gets deeper ($l \uparrow$), the value of $\hat{y}$ will explode.

Conversely,

$$w^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \rightarrow \hat{y} = w^{[l]} 0.5^{l} \cdot x$$

In this case, the activations end up decreasing exponentially.

Note:

$w^{[l]} > I$, $\uparrow$ exponentially

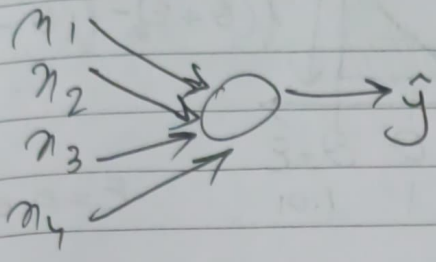
$w^{[l]} < I$, $\downarrow$ "

By similar arguments, the derivatives can also be shown to be increasing/decreasing exponentially.

So the right choices of weight matrices can make a lot of difference as if the ~~final value of output gets~~ gradient very small then the model will take a lot of time for gradient descent to learn anything as it will take small steps

3) Weight Initialization for Deep Networks

single Neuron example



$a = g(z)$

$z = w_1 x_1 + w_2 x_2 + \dots + x_n x_n + b$ (b → 0)

more the # of input feature, less should be the value of w_i as they are getting summed up but if w_i is large z will become very large

$Var(w_i) = \frac{1}{n}$ $\frac{2}{n}$ if activation f" is ReLU

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

→ This will make sure that $w^{[l]}$ is not bigger than 1 and also not very small than 1 so it doesn't explode or vanish very quickly.

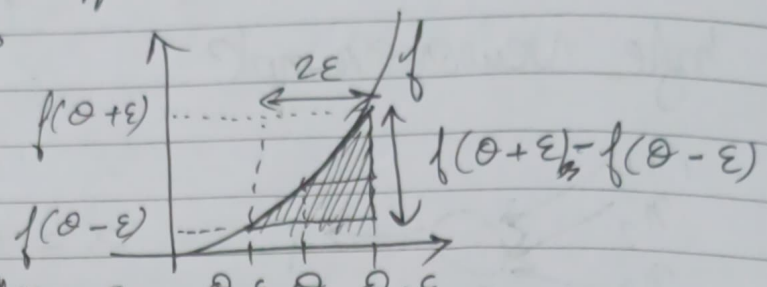
other variations:

tanh : use $\sqrt{\frac{1}{n^{[l-1]}}}$: (Xavier initialization)

or use $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

4) Numerical Approximation of Gradients

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



2 sided difference

$\varepsilon = 0.01$

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

$$\frac{1.01^3 - (0.99)^3}{2(0.01)} = 3.001$$

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

approx error : 0.0001

5) Gradient Checking

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

in assignment:
dictionary_to_vector() ← concatenate

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

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

gradients_to_vector ← concatenation

Is $d\theta$ the gradient of $J(\theta)$??

Grad check $J(\theta) = J(\theta_1, \theta_2, \theta_3, \ldots)$ upto dim of θ

for each i:

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

$$\approx d\theta^{[i]} = \frac{\partial J}{\partial \theta_i}$$

To check $d\theta_{approx} \overset{?}{\approx} d\theta$

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

$< 10^{-3}$

N^n: Euclidian distance

D^n turns this formula into a ratio

b) Implementing grad checking

* Doesn't work with dropouts.
or keep-prob = 1 while working with grad check