Course #2: Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

Week #1:- Practical aspects of Deep Learning.

↳ Setting up your Machine Learning Application:-

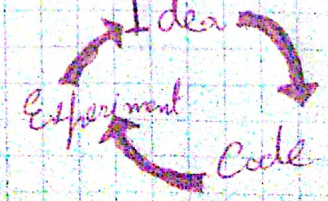
Train / dev / test tests:-

Now that we have learned that how to implement a deep neural network, now we will learn about the practical aspects of Machine learning i.e. how to make your Neural network work well by Counting in factors like hyper-parameter tuning, arranging your data or how to make your optimization algorithm run faster, so that you get you learning algorithm to learn in a reasonable time.

⇒ While training a neural network, you have to make a lot of decisions like, how many layers it has, how many hidden units you want it to have, what's the learning rate, and what activation function you are gonna use for different layers.....

Upon setting up you machine learning application, it is impossible to get all of there things right on your first attempt and that's why applied machine learning is an highly iterative process which follow the following diagram.

Loop →



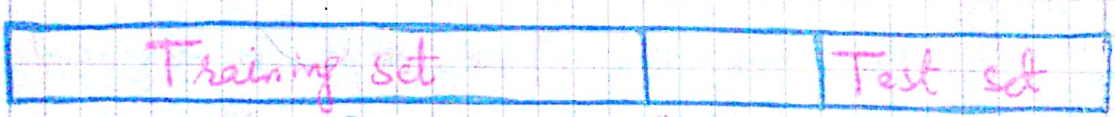
Today deep learning has found success in lot of areas like NLP, Computer vision, speech recognition even in structured data like Ads, web search, Computer security.

It often occurs that, intuition for the model from one area doesn't transfer to the other. Best choices depend upon the amount of data you have, no of inputs you have, Computer configuration. So, even experts find it very difficult to find values that are legitimate for our hyperparameters, that's why applied deep learning is very iterative process.

→ One thing that can make all this process optimized is that, how efficiently you can go around the loop and classifying your data into training, development and test efficiently can help you do that.

Traditionally you classify you dataset like this :-

You have some data :-



Data

Cross validation set / Development set ⇒ "dev"

You train your model on the training set, You go around the loop and try different models and test every model with Cross Validation set. After reaching to an ultimate model you finally test it on "Test set" and get an estimate that how well your algorithm is doing.

In previous era of machine learning the best practice was to split the data 70/30.
70% ⇒ training and 30% test or 60/20/20
60% ⇒ training, 20% ⇒ dev, 20% ⇒ test.
Previously we had not so large amounts of data, but in the modern era of big data where you have millions of data at your disposal, your dev and test set have become very small.

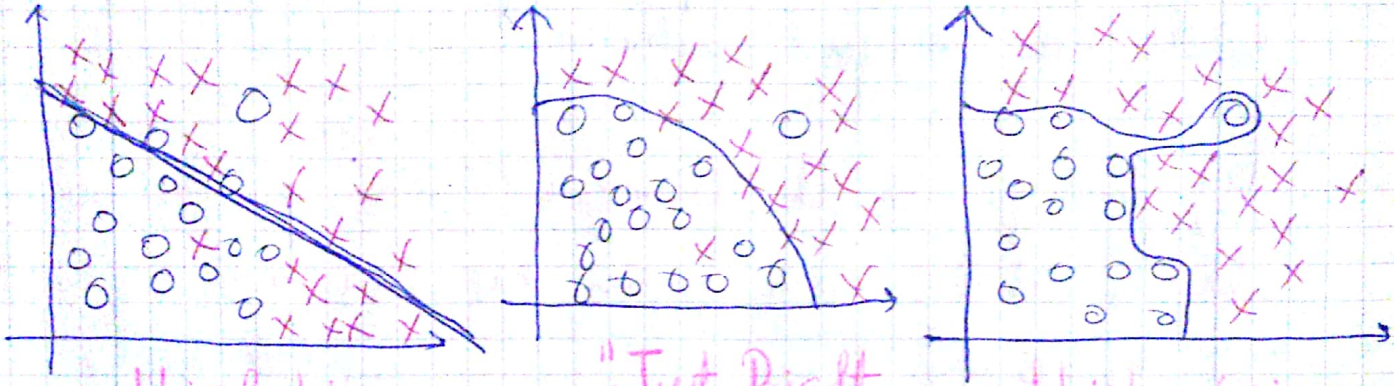
→ Mismatched train/test distribution:-
A lot of times it happens that, your train your data on the set gotten from one source and develop or test your data on a whole lot different set.
For example building a cat recognizing classifier you train the model on the set gotten from websearch but dev&tested test set is from users using your app or your own taken images. Now pictures from webpages would be of high resolution since professionals upload them there but your images in test and dev set can be blurry or not so good.
So, make sure your data i.e "all" of the data comes from the same source.

→ Not having a test set might be okay. (only dev set).

Bias / Variance :-

Bias and Variance are one of those sophisticated concepts that are easy to learn but difficult to master

In deep learning era we talk about bias, we talk about variance but we generally don't talk about bias - variance trade-off. Lets see what that means.



High bias
Under-fitting

"Just Right"

High variance
Over-fitting

→ In 2d problems you can plot the data and visualize bias and variance. In high dimensional data you can plot it and visualize a decision boundary. Instead there are couple other metrics that we'll look at in order to understand bias and variance.

→ Suppose you are dealing with image classification problem and you get following results :-

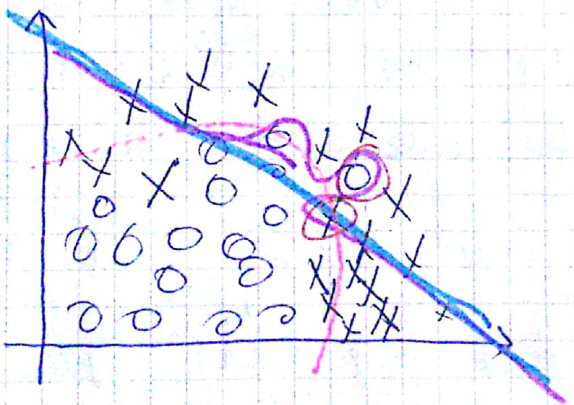
Train set error:	1 %	15 %	15 %	0.5 %
Dev set error:	11 %	16 %	30 %	1 %
	high variance	high bias	high bias & high variance	low bias / low variance

this analysis is done on the fact that human error is 0% i.e optimal (Bayes) error ≈0%

Suppose if your images are very blurry and even optimal error is as high as 15% then data in the second column of last table would not be classified as of high bias, infact it will be a good estimate.

By looking at the training set error you can tell that, whether your problem has high bias or low. and by assessing Dev set error you can tell about Variance

We visualized all the classifications that are showed in the last table but what does high bias and high variance looks like?



We said previously that linear classifier like this (shown in yellowish blue) has high bias because it underfits the data. But if somehow your classifier does some weird things like this (in purple) it will have high bias and high variance because it is overfitting the data.

Where it has high bias because by being a mostly linear classifier it's just not fitting the quadratic shape (in pink) very well, but by having too much flexibility in the middle it somehow overfits those two examples (in red).

So it has high bias because it was mostly linear and you needed a quadratic function and high variance because it has too much flexibility to fit those two middles in the middle.

Basic "recipe" for Machine Learning :-

When training a neural network, after having trained an initial, first ask, does your algorithm has high bias? look at the training data performance for this. If it does what you should do is try a bigger network or ~~more~~ more hidden units or train it longer or try better optimization algorithm. Keep doing this until you get rid of the high bias problem.

→ Once you reduce the bias then ask, do you have a high variance problem? To look at that, evaluate the dev set performance. If you have a high variance then best way to solve that is to get more data or you could try regularization or you can try different neural network architecture. Do these thing and see until you get a model with

low bias and low variance

→ Depending upon whether you have high bias or high variance, the set of things you should try could be quit different. I usually use performance of test and dev set to diagnose. So if you have high bias then getting more data is not the efficient thing to do.

In earlier era of machine learning that used to be a lot of talk about **bias Variance trade off**, because at times you reduce bias variance goes up and vice versa.

But in deep learning era training a deep network always reduces your bias without hurting your variance, as long as you regularize appropriately.

Regularizing your Neural Network :-

If your neural network is over-fitting your data i.e. high variance problem then one of the first things you should try probably is regularization. Other ways like getting more training data is also kind of reliable but you can't always get more training data.

Let's see how regularization works.

Let's develop this idea using logistic regression:-

Recall for logistic regression you try to minimize your cost function J i.e $\min_{w,b} J(w,b)$.

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

For regularization what you can do is add the term $\frac{\lambda}{2m}\|w\|_2^2$ in the cost function.

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

L2 regularization: $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

Now why do we regularize just the parameter w, why not b also? and add $(\lambda/2m) * \|b\|_2^2$. In practice you could do this but it won't make much of a difference. $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$. Almost all the features are in w and b is just one feature so it will not make any difference.

People also use L1 regularization: when the regularization term used is $\frac{\lambda}{2m}\sum_{i=1}^{n_x}|w_i| = 1\text{-norm}$.

If you use L1 regularization then w will end up being sparse.

Sparse means w will have a lot of zeros in it.

→ Some people think it can help compressing the model but it only help a bit not more hence not worth it.

One last detail :- λ here is regularization parameter. And you set this using your Cross-validation set. So λ is another hyperparameter that you'll have to tune.

How to implent regularization in a neural network?

→ In neural network you have a cost function over all your parameters.

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

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

"Frobenius norm"

Previously during back prop we were computing

$$dw^{[l]} = \underbrace{\text{from back prop}}_{\frac{\partial J}{\partial w^{[l]}}} + \frac{\lambda}{m} w^{[l]}$$

and perform update $\boxed{w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}}$

Now you add (after regularization) $\frac{\lambda}{m} w^{[l]}$

If we plug the $dw^{[l]}$ in the above equation then

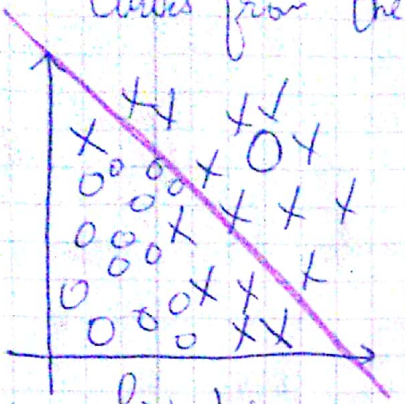
$$w^{[l]} := w^{[l]} - \alpha\left[(\text{from back prop}) + \frac{\lambda}{m} w^{[l]}\right]$$

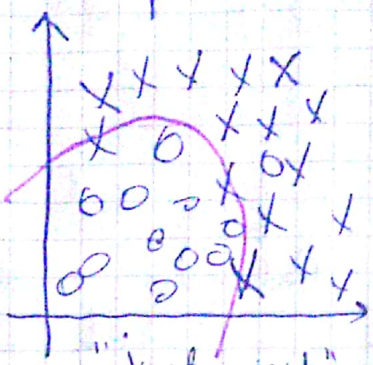
$$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha(\text{from back prop})$$

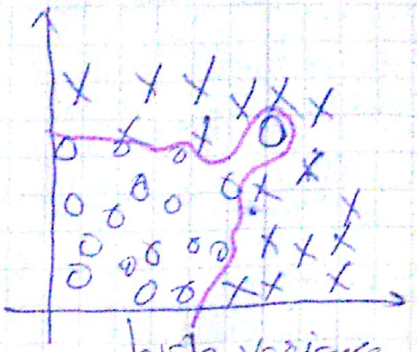
as $\left(1 - \frac{\alpha\lambda}{m}\right)$ is being multiplied by $w^{[l]}$ so it will decrease overtime. That's why it is called Weight decay.

Why regularization reduces overfitting?

- Recall the high bias, high variance and "just right" curves from the earlier part.







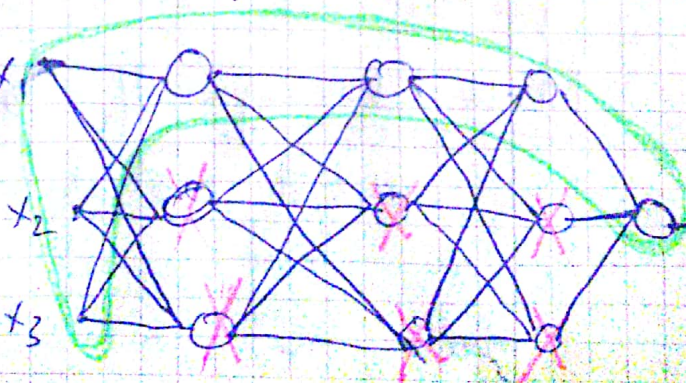
- let's think that you have some large neural network that is currently overfitting.

So Cost function:

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

So, why is it that shrinking the L2 norm or frobenius norm with the parameters might cause less overfitting?

- One piece of intuition is that, if you crank the regularization term (λ) to be really really big then it will incentivize the weight matrix W to be really really close to zero. So this means that it will zeroing out the impact of the lot of hidden units.



as shown in Pink.

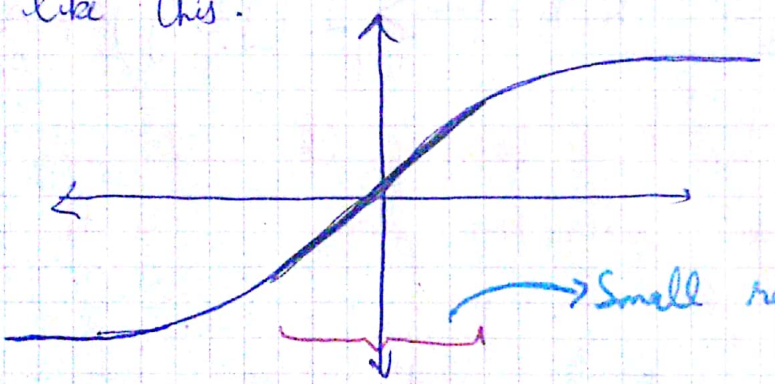
- It's almost like a logistic regression unit but stacked multiple layer deep

This will take you from the over fitting case to over-bias case. But hopefully there will be intermediate value of λ that will result our model to be "Just right".

Intuition about zeroing out hidden units isn't quite right. They are still there computing and all but they have much smaller effect. But effect regularization over network act as a simple on which is less prone to ~~Here is~~ over fitting.

Here is another attempt at understanding that, why regularization prevent over fitting?

Assume we are using tanh activation function which looks like this.



So, if z assumes a smallish value then we will wander around the linear region on the curve shown by black line, only if z becomes large in either directions the activation function start to become less linear

Main takeaway:- If regularization term λ is large then weight matrix would be small and if W is very small the z would also be small and in small region $g(z)$ will be

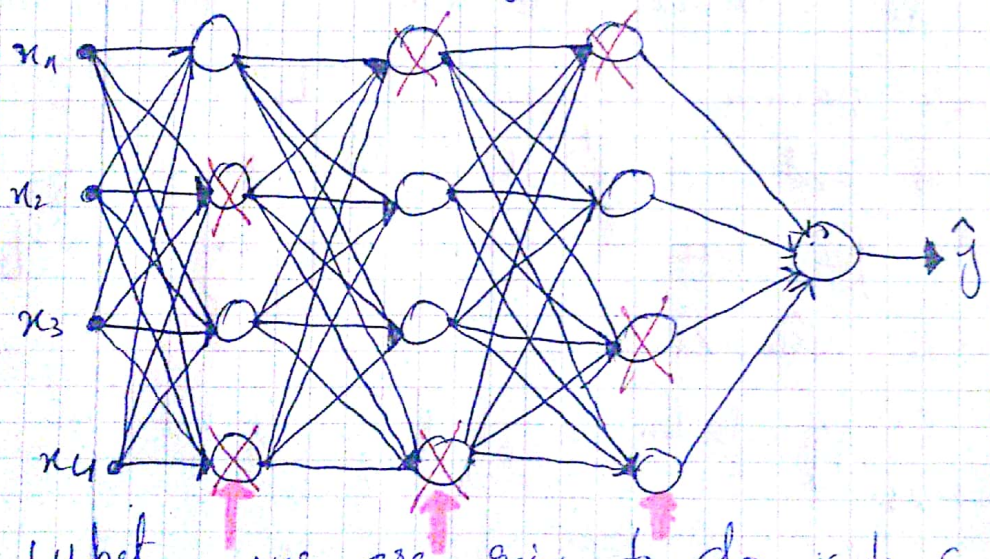
roughly linear and this suggest that every layer will act as roughly linear as if it is linear regression.

In the previous course we established that if the whole network is linear then even deep network will not be able to fit very very complicated decision boundaries that allow it to overfit the dataset.

Dropout Regularization:-

In addition to L2 regularization there is another very powerful technique called dropout. Let's see how that works.

Consider the following network to be over fitting.

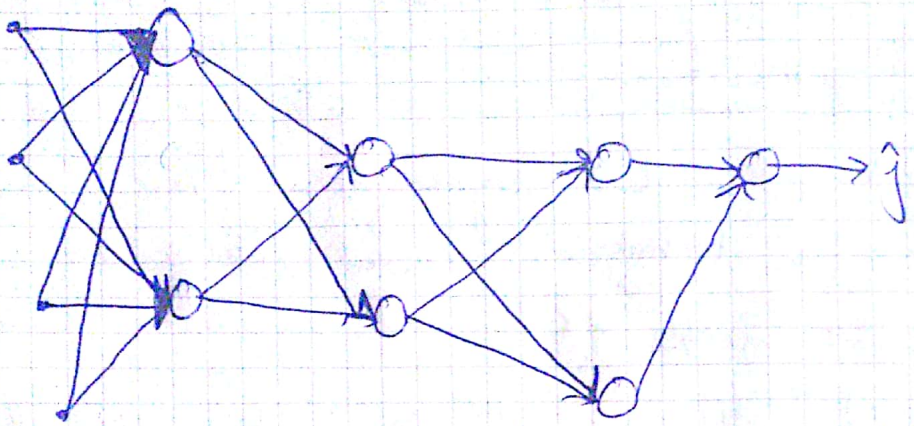


What we are going to do is to go through each of the layer of the neural network and set some probability of eliminating a node in a neural network.

Let's say for each of the layer we are gonna toss a coin and have 0.5 chance of keeping each node and same for removing each node.

and after coin tosses may be we decide to eliminate those nodes and what you do is remove all the outgoing and incoming connections from that node.

e.g. after coin tosses nodes marked in red are to be crossed out.



So this is the network we will be left with.

So, for each training examples you chose different set of nodes and cross out different set of nodes and you would train it using one of these neural networks.

Seems like crazy technique, but guess what, it actually works.

Let's look at how to implement dropout?

Let's look at the most common technique called "Inverted" dropout.

Let's illustrate this with a neural network of layer $L = 3$. keep_prob $= 0.8$ → so 80% remaining then and 20% chance of dropping out

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

a3 = np.multiply(a3, d3) → elementwise

d3 will be boolean array.

then we divide a3 by keep_prob → a3 = np.divide(a3, keep_prob)

What's this final line is doing here

→ Let's say for the sake of argument we have 50 neurons in a hidden unit. So may be for one example $a^3 \in \mathbb{R}^{50 \times 1}$ so with 80% chance of keeping them and 20% chance of eliminating them then on average you end up with 10 neuron dropped out.

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

↖ reduced by 20%.

In order to prevent the value of $z^{[4]}$ from dropping you divide it by 0.8 and this will bump it back up by 20%. So a3's expected value is not changed.

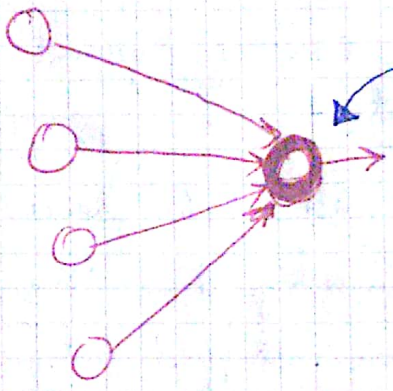
So during testing of your model this green encircled line makes test time easier, because you have less of a scaling problem.

Understanding dropout :-

One thing to remember is that, at test time you don't implement dropout because you don't want your prediction to be random.

Dropout does this seemingly crazy thing of randomly dropping neurons. Let's understand why that works out.

Let's look at from a perspective of a single unit. Say this one



For this unit to do its job it needs 4 inputs and needs to generate some meaningful output.

Now with dropout inputs can get randomly eliminated. So it means that, bold unit can't relay on any one feature, because any one feature or any one of its input can go away at random, so i'd be reluctant to put too much weight on any one of the input because it could go away.

So bold unit would be better off spreading out its weight to each of the four inputs

By spreading out the weights, this will tend to have an effect of shrinking the squared norm of weights.

So, this way dropout shrinks the weight and does the same as L2 regularization that helps prevent over-fitting.

It can be shown as an adaptive form of L2-regularization but L2 penalty on different weights are different.

One more detail:-

Suppose a neural network with 3 features, two 7 unit layers, then one 3 units layer, one 2 unit layer and then one unit layer

One of the ~~Chance~~ parameter that we have to choose is keep-prop which is the chance of keeping one unit or ~~and~~ kicking it out.

In this network the place where you worry more about overfitting like in layer 3 where $w^{[3]} \in \mathbb{R}^{7\times 7}$ you should keep the keep-prop to be low and bump up it's value where you might worry less about overfitting. The layer where you don't worry about overfitting at all, you can keep the keep-prop's value to be 1.

So keep-prop for different layers can be different.

So where there is a chance of more overfitting reduce keep-prop, kind of like cranking up λ in L2-regularization.

- Technically you can also apply dropout to the input layer where you sometime ~~add~~ drop out one or more features. In practice it is not done that way.

The downside of all this is that, it gives you more hyperparameters to work with.

One big downside:- Your cost function is not very well defined. On every iteration you randomly killing off bunch of nodes.

Other Regularization Methods:-

Data Augmentation:

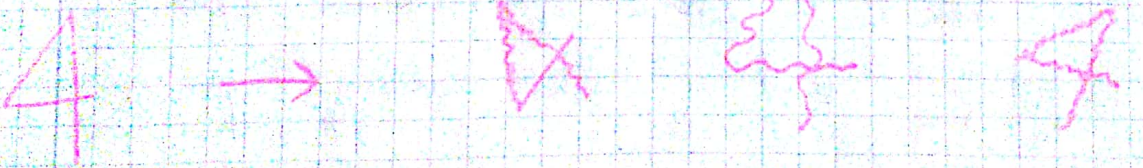
If you are training a cat classifier and it is over-fitting, getting more training data can help but it can be expensive, what you can do is take an image and flip it horizontally and add it to your dataset.

This way you can double the size of training set. This will not be as good as getting more m training examples but it certainly will save the cost.

Other then rotating horizontally you can also rotate the image and zoom it in to a certain degree.

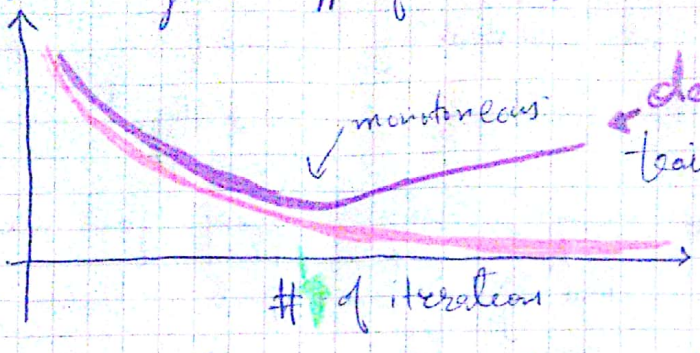
In character recognition like numbers you can alter the shape of numbers like

From → To



Early stopping :-

- When you are training a neural network you plot cost function against # of iterations



- What you should do is also plot dev set error. Now you see it comes down to some extent then the error goes up. What you would do is the place where dev set error stopped decreasing is the place where you stop training your model.

Now why this works?

When you start iteration $w \approx 0$ (due to random initialization) but as the cost decreases it increases and at the end we end up with large w.

- So what early stopping does is that, by stopping half way you have only mid-sized $\|w\|_F^2$ and Similar to L2-regularization by small norm w your model will prone to overfitting less.

It does have one downside :-

In machine learning upon training a model what you do is optimize a cost function J by using a legible optimization algorithm

- The next thing you see after training is access whether your model is overfitted or not then

If it is, you try to regularize it. So these 2 are totally different things and in early stopping you cut-hold the aforementioned 2 tasks i.e. optimizing and eliminating over-fitting. You cannot work on the problems independently.

→ Rather than using early stopping you can use L2-regularization, then you can just train the network as long as possible, it's downside is that you have to try different values of λ which is computationally expensive

Setting Up Your Optimization Problem:-

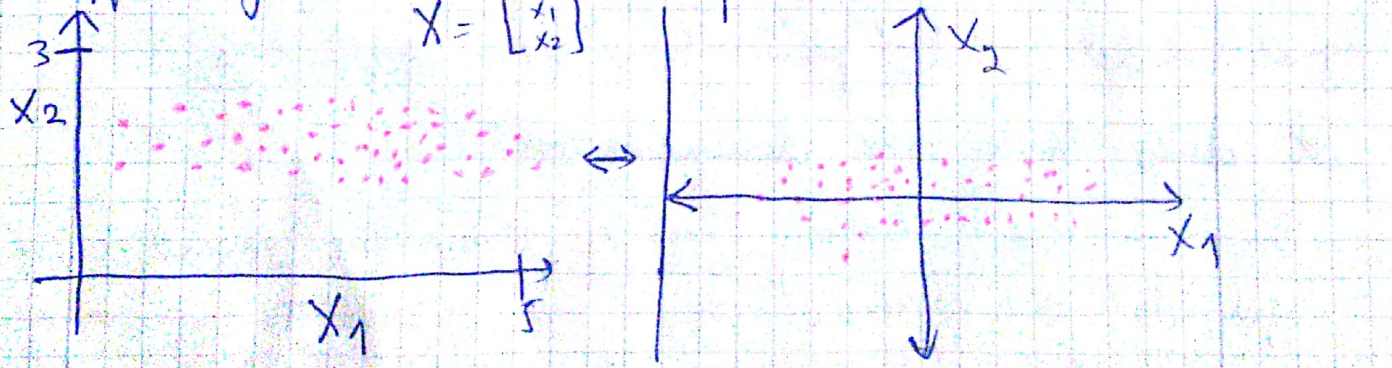
Normalizing Inputs:-

- One of the techniques that will speed up the process of training your neural network is normalizing your inputs.

That goes the following way:

Suppose you have 2-d input

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



Original shape

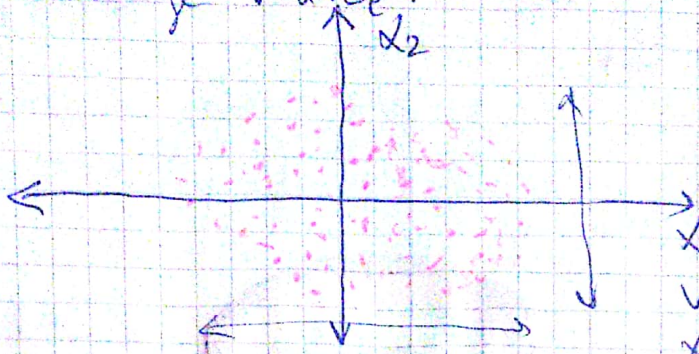
Subtract mean

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

$$x := x - \mu$$

Mean of the whole data set is zero

Next you normalize variance.



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

$$X /= \sigma^2$$

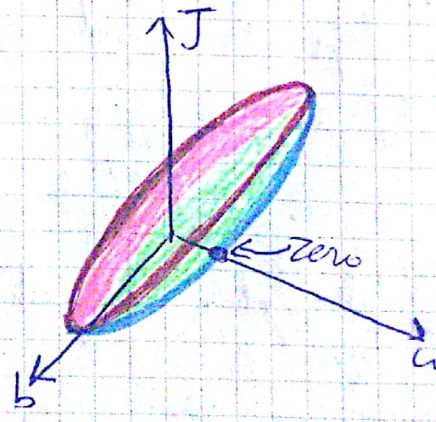
Variance of both x_1 and x_2 is equal to 1.

→ Use the same μ and σ^2 to normalize your test set also, because you want your data to go through same transformation.

Now question is, why do we want to normalize input features.

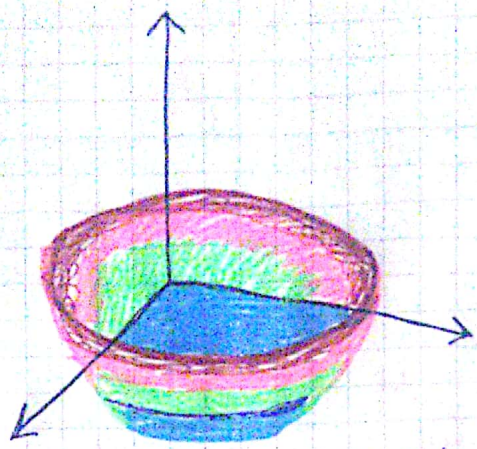
Recall the cost function $\boxed{J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}$

If you use unnormalized data, your cost function can assume an elongated shape. e.g x_1: 1 .... 1000 and x_2: 0, ... 1 then we get graph like this.

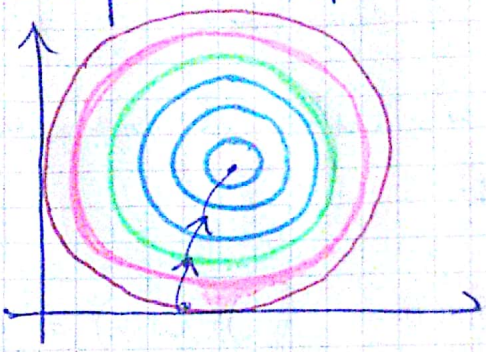


and its contours are like this

but if we use normalized data we get graph like this



and it's contour plot looks like



Gradient descent will have trouble finding zero in Unnormalized data. ib shown in the lost fig whereas in normalized data gradient descent can even use larger steps to find ~~gradient descent~~ zero.

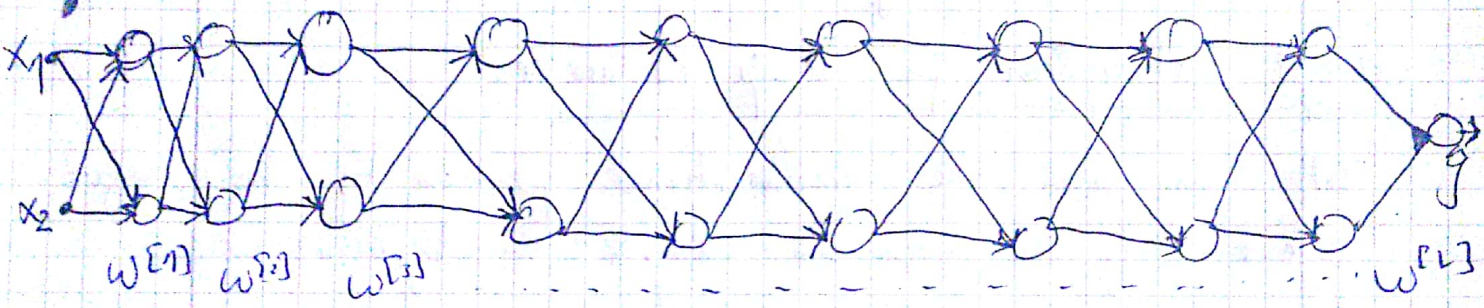
So, if there is very much high difference in ranges of the features it is better to normalize the data because optimization algorithm works better that way.

Vanishing / exploding Gradients :-

- When training a neural network one of the problems you encounter is either your gradients get very high or they can get to zero and this thing make the learning very very slow.

→ Let's see how careful random weights and bias initialization can help us solve this problem.

Let's say you have this very deep hidden network.



For the sake of simplicity let's say you have linear activation function i.e $g(z)=z$ and all the biases are zero.

In that case your output will be:

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

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

$$w^{[2]} z^{[1]} = a^{[2]}$$

Let's say:

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

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

So: $\boxed{\hat{y} = [1.5]^{L-1} X}$ So if you have a very deep neural network i.e $L \gg 1$ then value of $\hat{y}$ will explode. Conversely if $W^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$ then

$\hat{y} = [0.5]^{(L-1)} X$ and if $L \gg 1$ then $\hat{y}$ will diminish exponentially as a function of no. of layers.

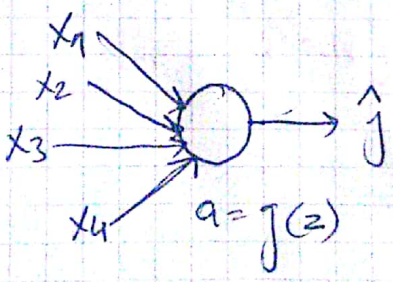
→ There is a solution that partially solve this problem, to see that, let's go on to that.

Weight initialization for Deep Networks:-

We can solve the aforementioned problem by the careful choice of our random weights initialization.

→ To understand this let's see to initialize the weights for a single neuron then we will go to generalize this concept for the whole network.

Single neuron example:-



$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$

$b = 0$

larger n → smaller w_i

If you have lot of units then you want smaller w_i's

One reasonable thing to do is to set

$$Var(w_i) = \frac{1}{n}$$

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

If you are using a Relu function then $Var(w_i) = 2/n$ works better.

→ The reason I have used $n^{[l-1]}$ because in generalized way we have features of layer $l-1$.

Other Versions:- If you are using tanh function then use the following formula:-

$$\sqrt{1/n^{[l-1]}}$$

For this, there is another initialization:-

Xavier initialization:-

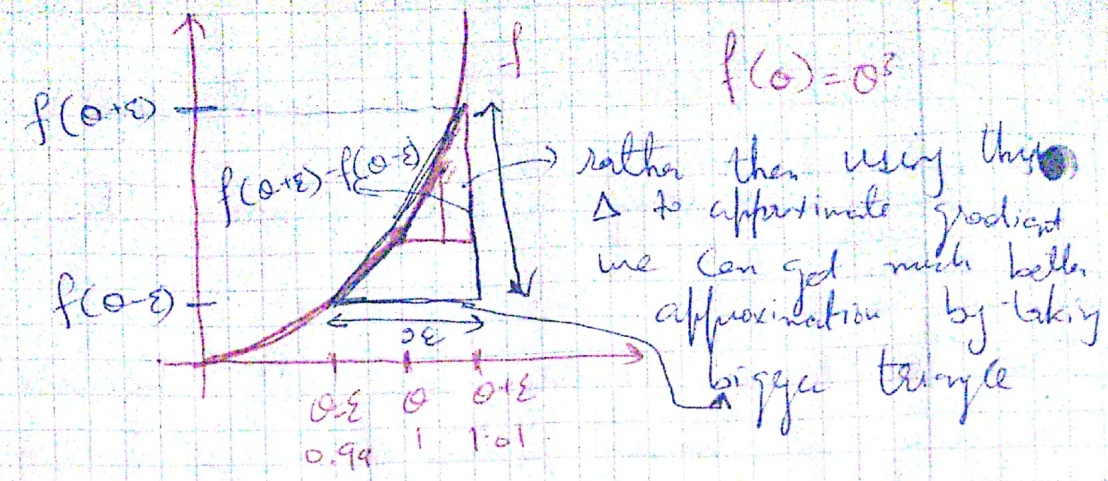
$$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$$

You can also count it as one of the hyperparameters and tune it accordingly.

Numerical approximation of Gradients:-

When you implement backward propagation, there is a test called gradient checking which can really help you to access that, your implementation is correct.

For this let's first see how gradients are numerically approximated.



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

rather than using this Δ to approximate gradient we can get much better approximation by taking bigger triangle

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

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

which is very close to 3

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

→ Center difference approximation

$\Rightarrow O(\varepsilon^2)$

Gradient checking :-

→ Used to debug the implementation of gradient descent.

To implement gradient checking what you should do is take all your parameters $W^{[1]}, b^{[1]}, \ldots W^{[l]}, b^{[l]}$ and reshape them into a big vector θ.

Now instead of $J(W^1, b^1, \ldots W^l, b^l)$ you have

$J(\theta)$

Now take $dW^{[1]}, db^{[1]}, \ldots dW^{[l]}, db^{[l]}$ and reshape it into a vector $d\theta$.

Now the question is that, is $d\theta$ the gradient of $J(\theta)$?

Now here is how you implement gradient checking.

keeping in mind J is a function of θ

$$J(\theta) = J(\theta_1, \theta_2, \theta_3 \cdots)$$

for each i:

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

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

$d\theta_{approx}$ is same as $d\theta$ in dimensions.

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

Check $\dfrac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$ } if this is $\approx 10^{-7}$ then great if 10^{-5} might be ok but double check the parameters if 10^{-3} then worry.

Gradient Checking Implementation notes:-

- Don't use grad check in training - only to debug.
- If the algorithm fails grad check, look at components to try to identify bugs.

-what this means is that, $d\theta_{approx}$ is very far from $d\theta$ then you look at different values of (i) to check which of $d\theta_{approx}$ are far from $d\theta$.

- Check corresponding $db^{[l]}$ and $dw^{[l]}$ and look if there is fault in their implementation.

- Remember regularization.
- Grad check doesn't work with dropout.
  What you can do is do grad check without dropout and then turn on Dropout.
- Run at random initialization; perhaps after some training.