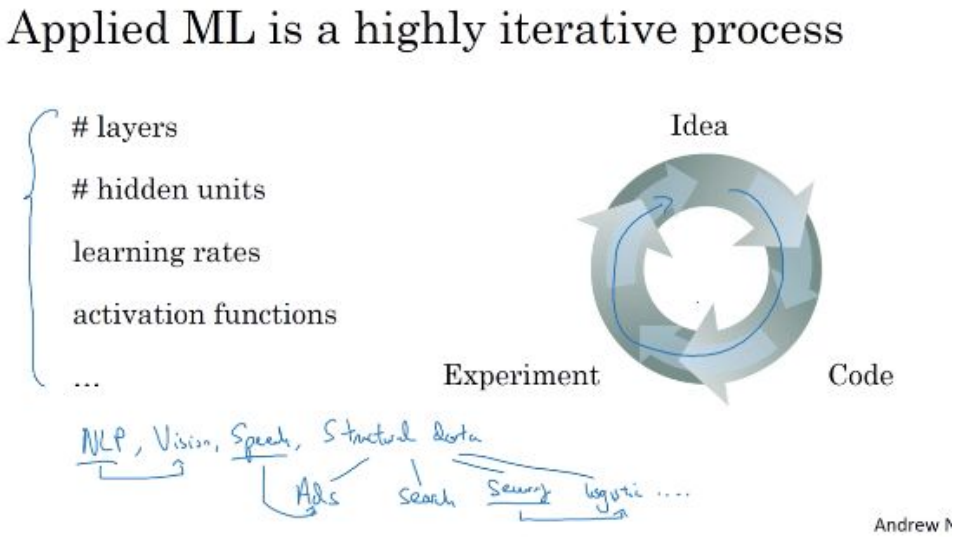
Andrew Ng's Improving Deep Neural Networks Class

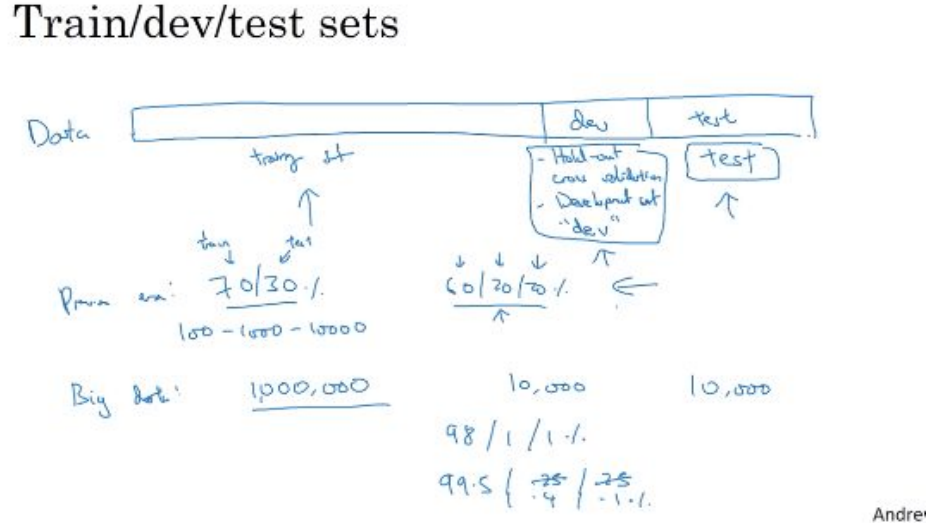
Week 1: Practical Aspects of Deep Learning

Setting up your Machine Learning Application: Train/Dev/Test Sets

- Applied Machine Learning is a very iterative process. You need to decide on a number of parameters and it's not clear at the beginning how to decide. Furthermore, ML is used in many applications (NLP, Vision, Speech, Ads, Search, Security, Logistics, etc.) and the intuitions learned in one vertical often do not translate to another vertical. So what you have to do is go through the cycle many times to find a good choice for your network.

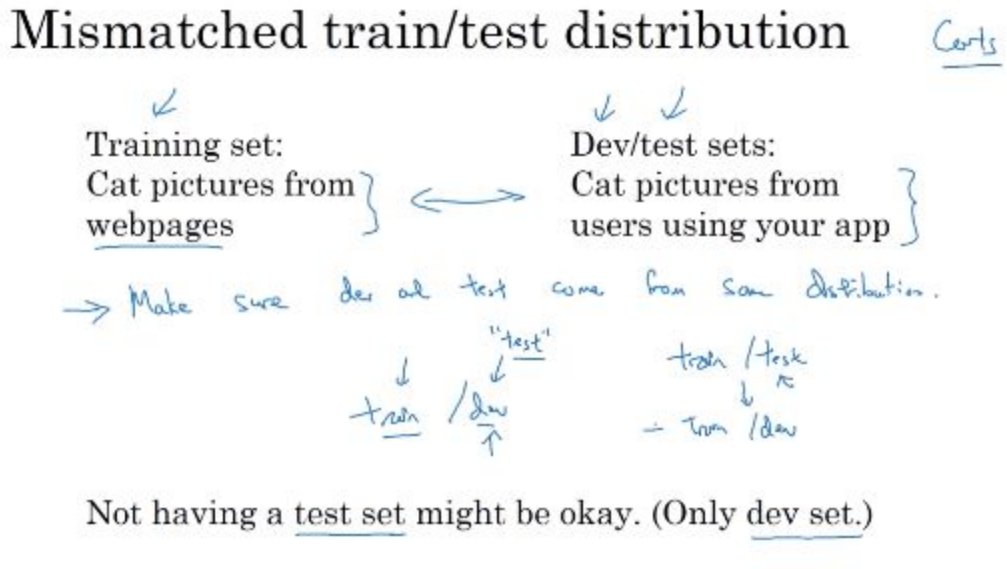


- Since you have to go through the cycle many times you should aim to go through it efficiently. One of the things that help efficiency is setting up well our datasets.



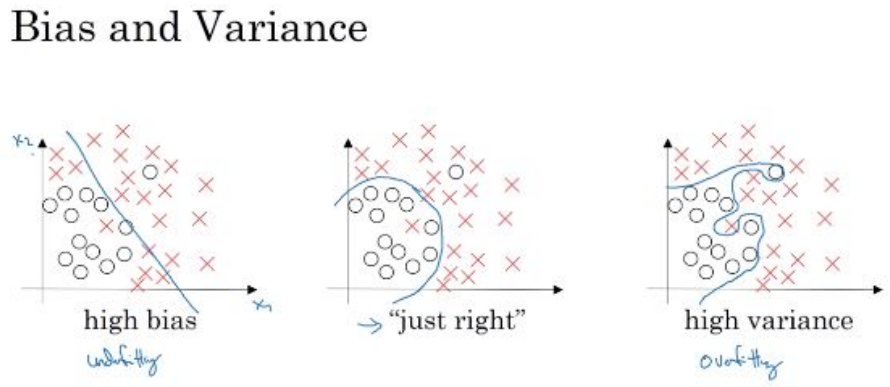
We typically have 3 sets: the training set, the dev set and the test set. We train our algorithms in the training set. We test them out on the dev set to see which one performs better and finally when you decided on a final model you use test that model in the test set. In an earlier ML era the split between the set types was 70/30 or 60/20/20. This was OK for test set sizes of 100 to 10,000. In the era of big data the splits we are seeing is more along the lines of 98/1/1 or 99.5/0.4/0.1.

- Another trend to be aware is people training on mismatched train/test distribution. What we mean by mismatched is that the train/test sets come from different places as shown in the example below. In this example, the training set comes from the internet and the dev & test sets come from an app. This is OK, the rule of thumb is to make sure the dev and test sets come from the same place/distribution. In some cases we might only have a train and a dev test because sometimes you might not need an unbiased estimate of the performance of the selected network. If you don't need this unbiased estimate it's on to only have 2 sets: train and dev (in practice we hear people calling them train and test)



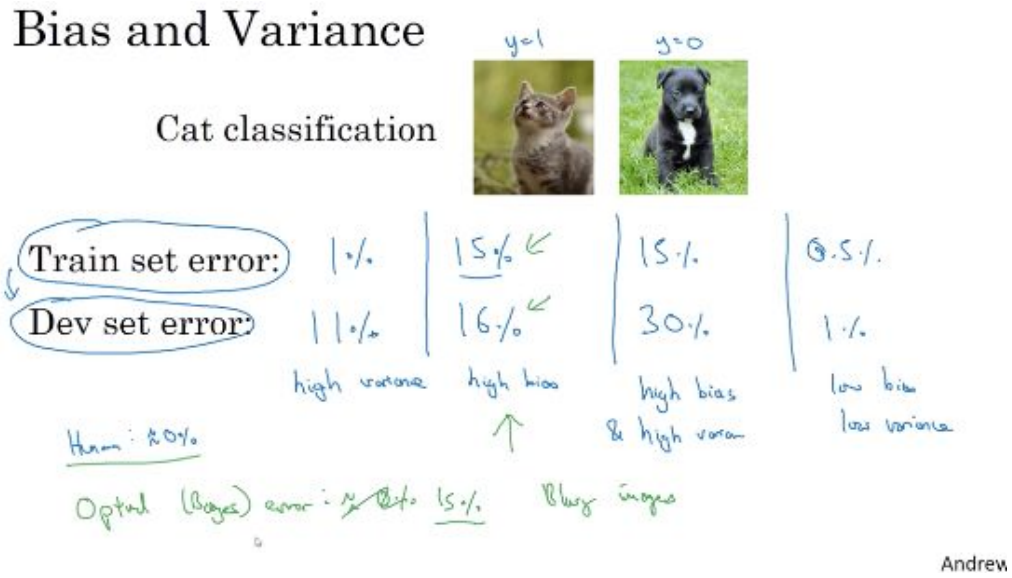
Bias/Variance

- Below we have a 2-d plot to illustrate bias and variance.



If we are underfitting the data we can say we have high bias (meaning our model/equation is too simple and we would be better served by using a more complex equation/model). If we are overfitting the data we can say we have high variance (meaning we are trying too hard to come up with a model that fits this dataset but will not translate well to other data sets) and we also have a problem. Somewhere in the middle we have a model that is 'just right'.

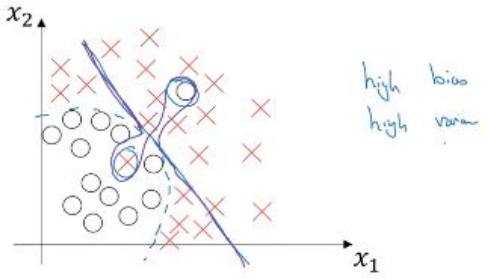
- Let's take a look at a couple of metrics that can help us understand bias and variance. These metrics are the train set error and the dev set error. Looking at the train set error can tell us if we have a bias problem (we are underfitting our dataset). Looking at the difference between the train set error and the dev set error can tell us if we have a variance error because the purpose of the train set is to generalize well to the dev set. If it's not generalizing well we will have a 'big' difference between the train set error and the dev set error.



The 4 examples above show: high variance (small train set error but big difference between train set error and dev set error), high bias (big train set error but small difference between train set error and dev set error), high bias & high variance (big train set error and big difference between train set error and dev set error) and finally, low bias and low variance (small train set error and small difference between train set error and dev set error). All of these examples are predicated on two things: the human error (also known as the optimal (Bayes) error is around 0% and both sets come from the same distribution/place.

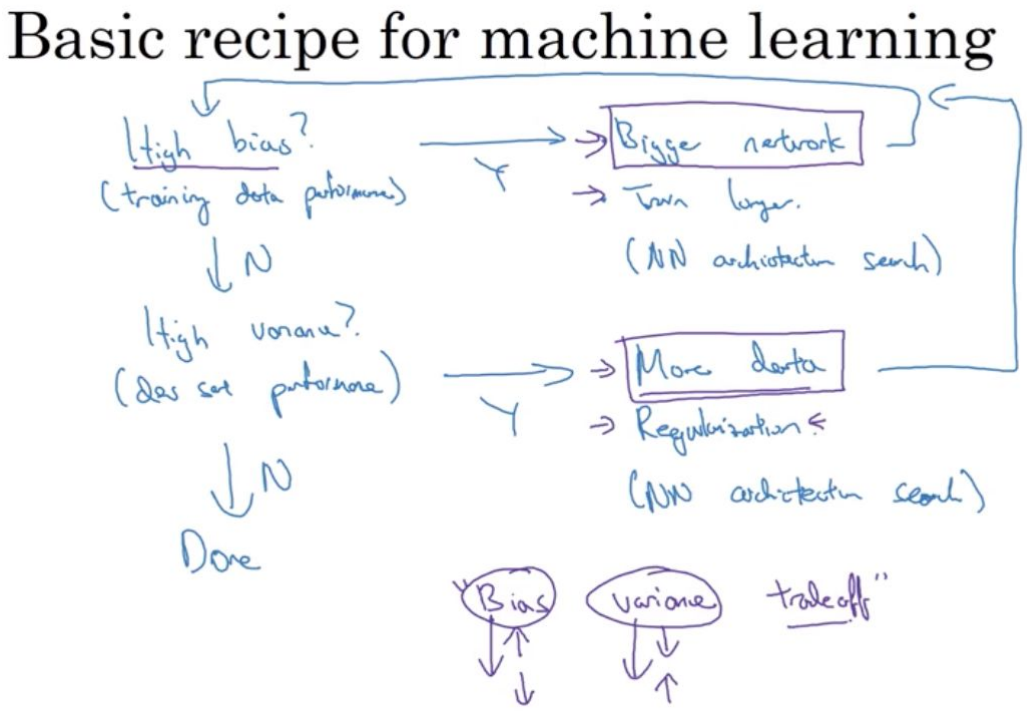
- How do we represent high bias & high variance? Let's take a look in a 2-d plot shown below. In this case we see an algorithm in purple that has some 'linear' parts where it is underfitting the data (high bias) and other parts where it is 'trying too hard' to fit the data (high variance).

High bias and high variance



Basic Recipe for Machine Learning

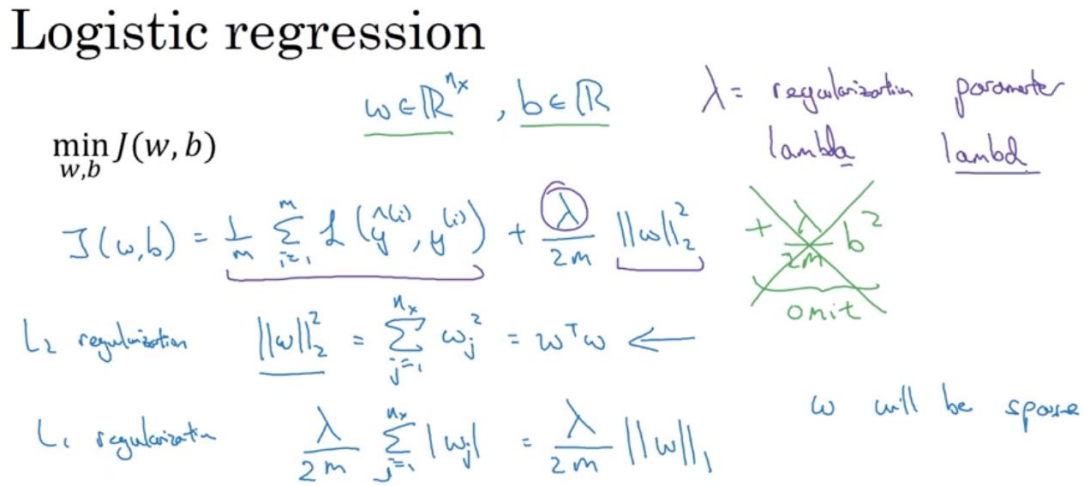
- A basic recipe to diagnose bias/variance and how to do something about it is shown below. The steps are
  - Ask if we have high bias by looking at the performance of the training data set
    - If Yes then we can try using a bigger network (i.e. more hidden layers), train the existing network longer or look for another NN architecture
    - If No we move on the next step
  - Ask if we have high variance by looking at the performance of the dev set.
    - If yes then we can try using more data, regularization or look for another NN architecture
    - If no we are done.



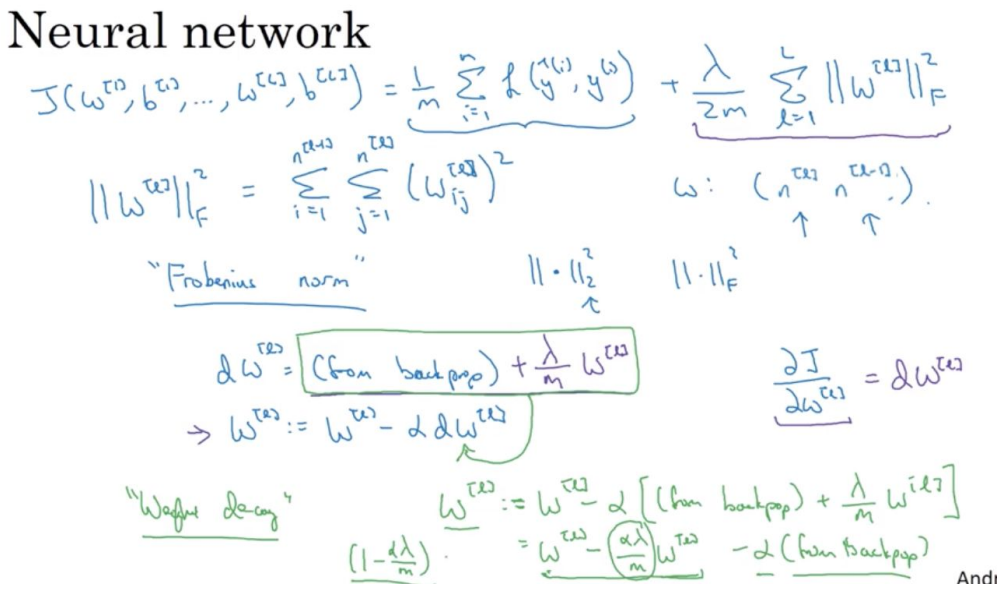
Before Deep Learning we talked more about the bias/variance tradeoff where minimizing one would increase the other. This tradeoff is now less of a problem thanks to our ability to get more data and/or to use a bigger neural network.

Regularizing your Neural Network: Regularization

- We are going to look at regularization because it is 'cheap' way to reduce overfitting/high variance. Let's take an initial look using logistic regression. To add regularization what we do is to add the regularization term that uses lambda (λ). Our rewritten cost function $J(w,b) = 1/m * \sum_{i=1}^{m} L(y^{*(i)}, y^{(i)}) + (\lambda/2m) * ||w||_2{}^2$ and $||w||_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ and this is called L2 regularization. We might also hear about L1 regularization which is defined as $(\lambda/2m) * \sum_{j=1}^{n_x} |w_j| = (\lambda/2m) * ||w||$. L1 will create sparse w vector (a vector with a lot of zeros in it) and some people say this helps in compressing the model because zeros take up less memory. However, Professor Ng says it is not used often and L2 is used way more.



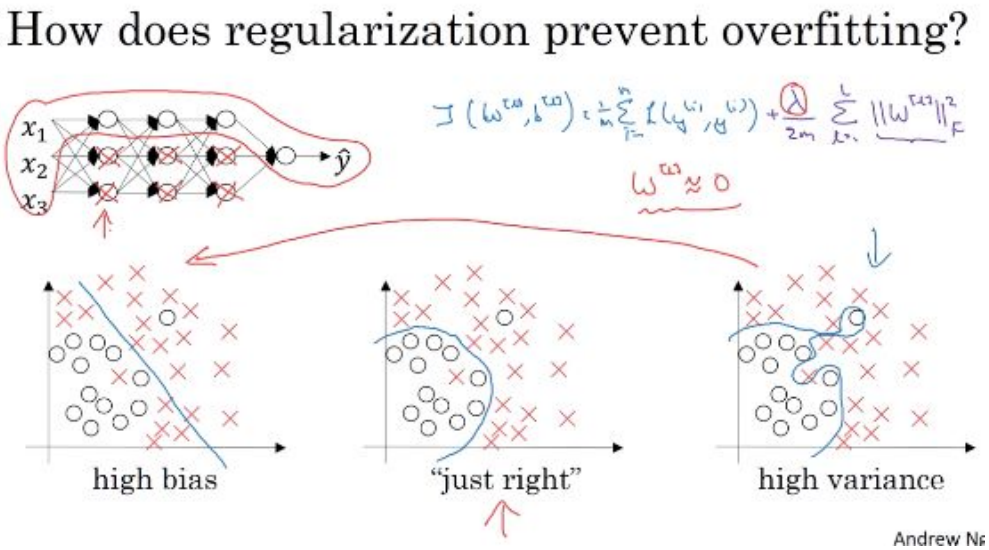
- Now let's see how we do regularization for a neural network. In a neural network we rewrite our cost function as follows:
$J(w^{[1]}, b^{[1]}, ..., w^{[L]}, b^{[L]}) = (1/m) * \sum_{i=1}^{m} L(y^{*(i)}, y^{(i)}) + \lambda/2m * \sum_{i=1}^{L} ||w^{[L]}||_F^2$ where
$||w^{[L]}||_F^2 = \sum_{i=1}^{n^{[L-1]}} \sum_{i=1}^{n^{[L]}} (w_{ij})^2$ and this term is called the Frobenius norm of a matrix and is just the sum of the squares of the elements of the matrix.



Given that, how do we implement gradient descent? What we do is we add another term to our dw computation, our rewritten dw is then $dw^{[l]} = (from\ backprop) + (\lambda/m) * w^{[l]}$ and we then update $w^{[l]}$ as follows: $w^{[l]} = w^{[l]} - \alpha * dw^{[l]}$. This regularization is also sometimes called weight decay because what it does is it reduces the matrix a little bit.

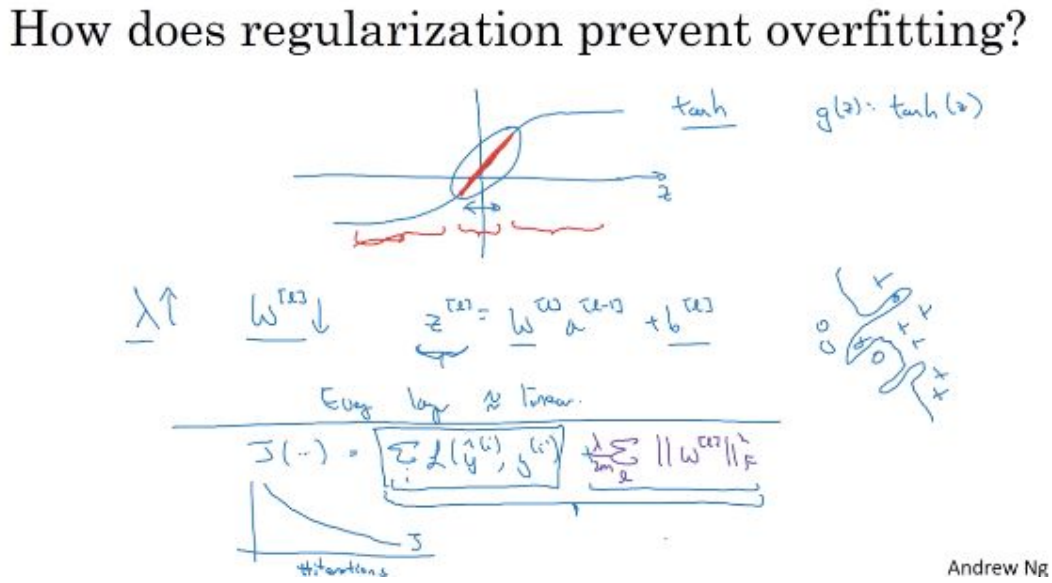
Regularizing your Neural Network: Why Regularization reduces overfitting?

- Now let's take a look why regularization prevents overfitting. Remember our common 3 cases that we have seen before: High bias, just right and high variance. Last time we defined a regularization term that is added to our original cost function. The new cost function is then $J(w,b) = 1/m * \sum_{i=1}^{m} L(y^{*(i)}, y^{(i)}) + (\lambda/2m) * ||w||_2^2$. Notice the lambda term.



What happens is the following: if we have a large λ we are effectively penalizing our w's and making $w \approx 0$. The effect of this is that our neural network is simplified because we are diminishing the impact of many hidden neurons and we end up with a 'simpler' neural network that moves away from high variance and towards 'just right' or maybe high bias. We are then reducing the chances of overfitting by adding the regularization term.

- Another intuition is the following: consider the $tanh$ activation function shown below. If we add a large regularization parameter (with a big λ) we are penalizing $w^{[l]}$ and therefore my $z^{[l]}$ will be small. If that is the case the $tanh$ value is within the linear part (shown in red below) of the $tanh$ function and since it is linear it will not be capable of doing complicated stuff like the one shown in the bottom right. It is, therefore, less prone to overfitting.

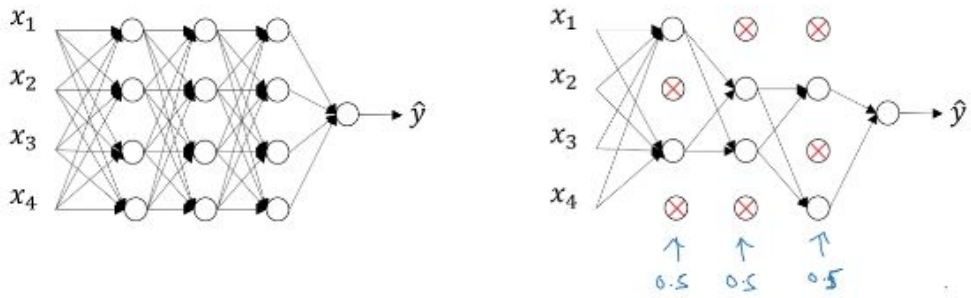


One last detail is that when we plot gradient descent over the # of iterations to see if it is decreasing, we need to include the regularization term. Otherwise, we might not see a constant decrease.

Dropout Regularization

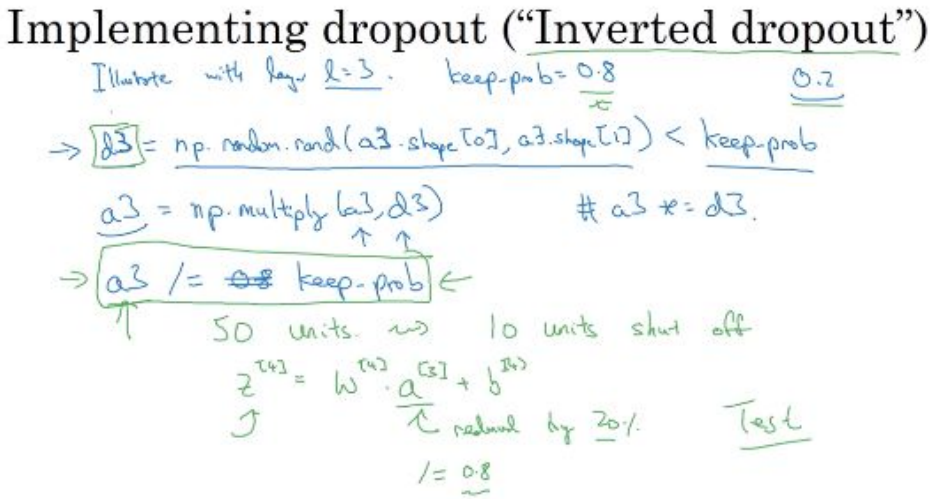
- Another powerful regularization method is dropout. Let's see how it works. What we do in this method is the following: we assign a probability (like flipping a coin, say 0.5) to each node in the network and then we remove some of those nodes. We then end up with a smaller/simpler network and do backprop on that network. In other examples, we eliminate different nodes and we end up with a different network so each training example is using a different neural network. This helps with overfitting because you have, at the end, a simpler, smaller network.

Dropout regularization

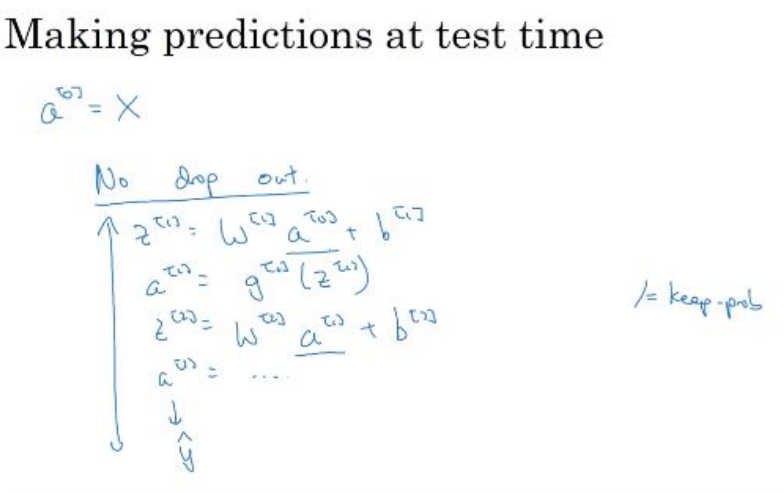


- How do we implement dropout? The most common implementation is inverted dropout and works as follows:
  - We have a vector $d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep.prob$
  - We then do $a3 = np.multiply(a3, d3)$ to zero out the selected neurons
  - Finally, we divide $a3/keep.prob$

  The last step is the inverted dropout and we do it to ensure $a3$ remains the same. If not, the $a3$ vector will be reduced by some margin (20% in the example below) and when multiplying it by $w^{[4]}$ it will reduce the value of $z^{[4]}$ by 20% and we don't want that.

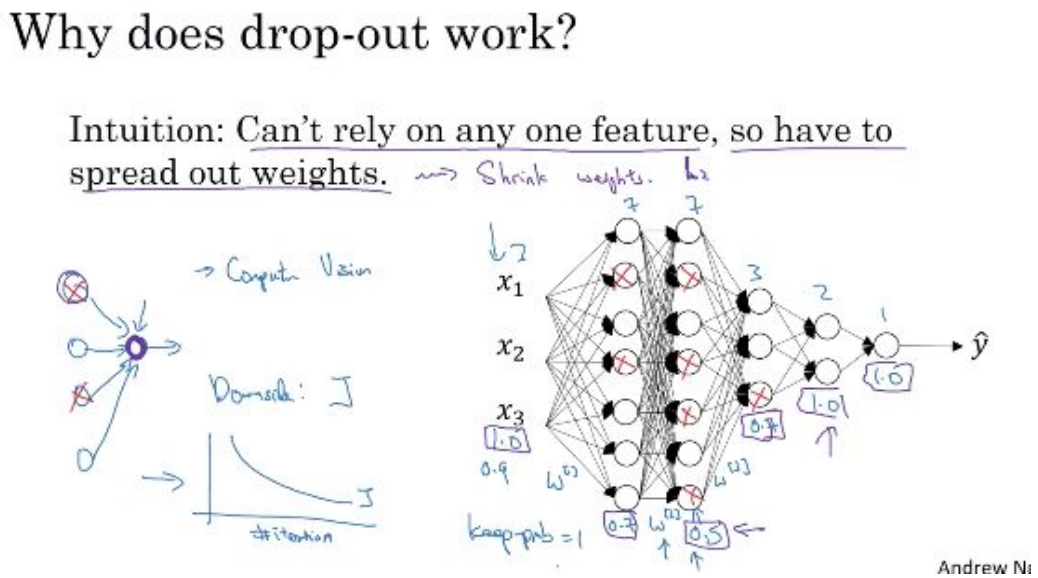


- What do we do at test time? At test time we don't use dropout regularization because it just adds noise at this step and we don't want that during the test phase.



Understanding Dropout

- Why does dropout work? Let's take a look at it from the point of view of one node, like the one on the left. As we know, dropout will randomly eliminate some inputs every iteration so the node cannot rely on any one input so it has to spread out the weights. It has to do this because it cannot count on any feature to be present all the time. Therefore, it shrinking the squared norm of the weights.

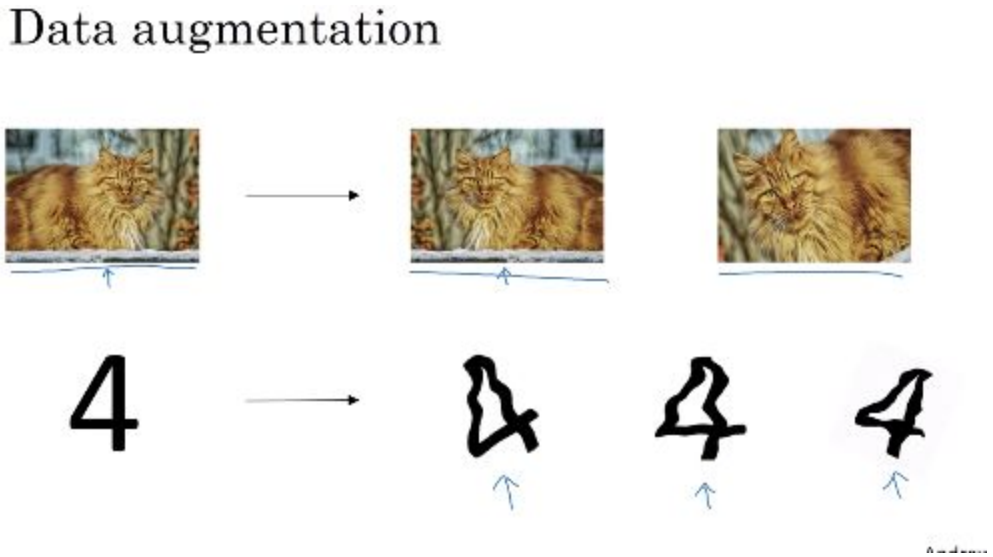


One more detail is that for every layer on a neural network I can have a different $keep.prob$. If I have many nodes in a layer my $keep.prob$ might be equal to say, 0.5 because I am worried this layer is overfitting. If I only have a couple of nodes in one layer it is less likely that I am overfitting and therefore my $keep.prob$ will be high, 0.9 or 1.0. The downside of this approach is that we have more hyper parameters to search for and we also have a less defined cost function because it changes all the time or, better said, every iteration. To overcome that fact what Professor Ng recommends is we run our

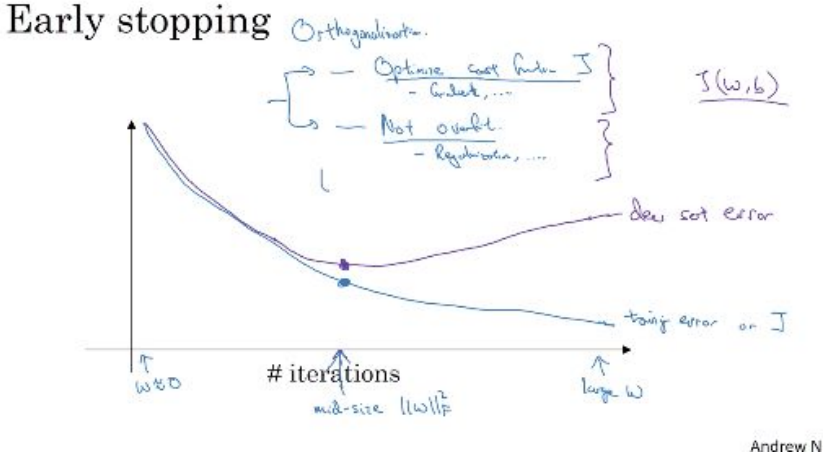
code without dropout to make sure the cost function is decreasing every time we are doing gradient descent. Once we confirm that we can turn on dropout and see how our process goes.

Other Regularization Methods

- We are going to see other regularization methods. The first one is data augmentation. Data augmentation is cheaper and faster than going out and collecting more data. In the case of images we flip them horizontally or take random crops of the original image. In the case of optical character recognition we can take the original character (a digit in this case) and apply random distortions and notations to it.



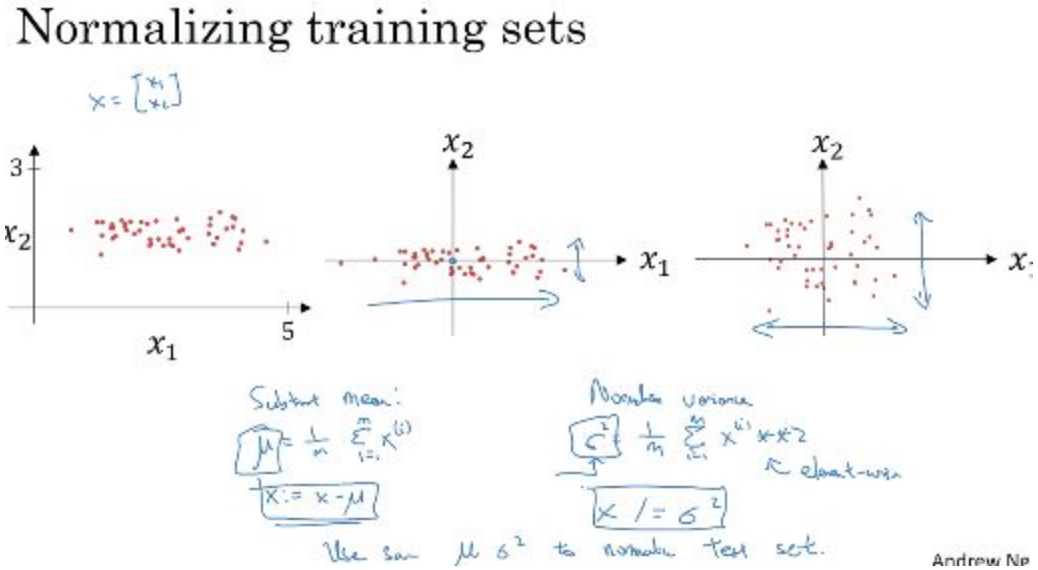
- Another technique used is early stopping. With early stopping we graph the cost function J and we also graph the dev set error. What we find most of the time is that the dev set error will decrease and eventually start increasing again. Early stopping tells us to stop before the dev test error starts increasing again.



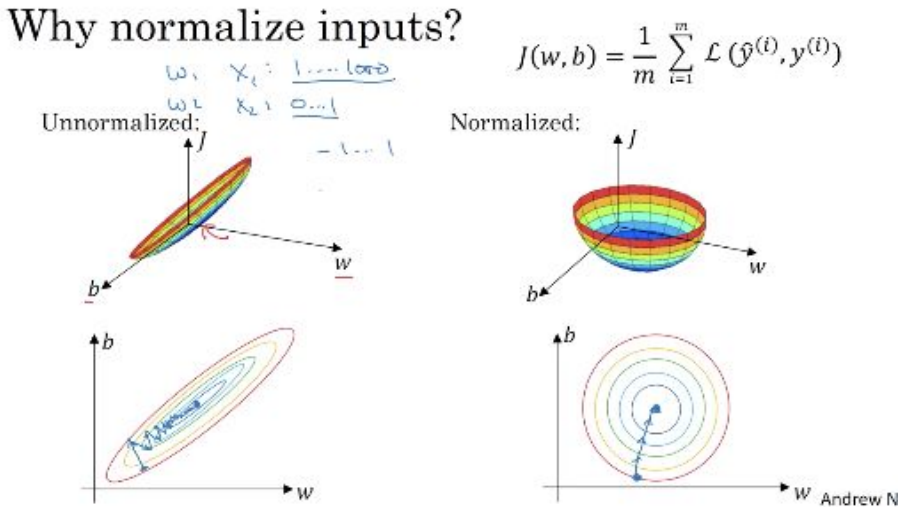
One disadvantage of this method is that it couples the tasks of minimizing the cost function (finding parameters w and b) and of preventing overfitting. The principle of separating these 2 tasks is called orthogonalization.

Setting up the Optimization Problem: Normalizing Inputs

- Normalizing your inputs means two things: you subtract the mean and then you normalize the variances. Subtracting the mean means having $\mu = 1/m * \sum_{i=1}^{m} x^{(i)}$ and then subtracting μ from X as $X = X - \mu$. The second step is to set $\sigma = 1/m * \sum_{i=1}^{m} x^{(i)} ** 2\ (element-wise\ squared)$. Finally we should use the same μ and the same σ for both the training set and the test set.



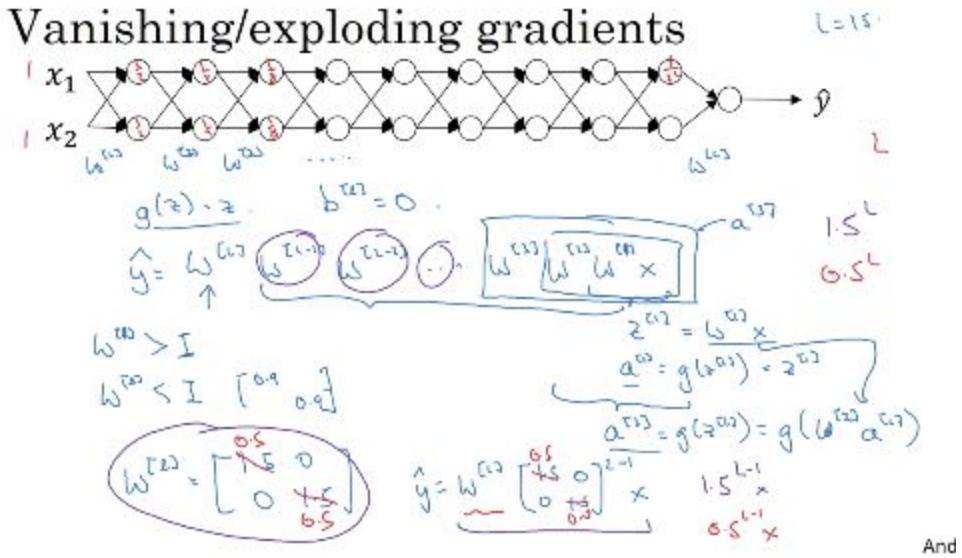
- Why do we normalize our inputs?



Because it makes our cost function J easier and faster to optimize. If we have unnormalized inputs and the inputs have dramatically different scale we end up with an elongated cost function like the one on the left above. Our learning rate will be small and gradient descent will need many steps to converge. A normalized cost function looks like the one on the right. In this case gradient descent can take larger steps and converge faster.

Setting up the Optimization Problem: Vanishing/Exploding Gradients

- One problem of neural networks (especially deep neural networks) is exploding and vanishing gradients. In the neural network below we have weights $w^{[1]}, w^{[2]}, ..., w^{[L]}$ and our $yhat = w^{[L]} * w^{[L-1]} * w^{[L-2]} * \ldots * w^{[3]} * w^{[2]} * w^{[1]} * X$ where $z^{[1]} = w^{[1]}X \; and \; a^{[1]} = g(z^{[1]})$ and $a^{[2]} = g(z^{[2]}) = g(w^{[2]}a^{[1]})$ and so on until we get to $yhat$.

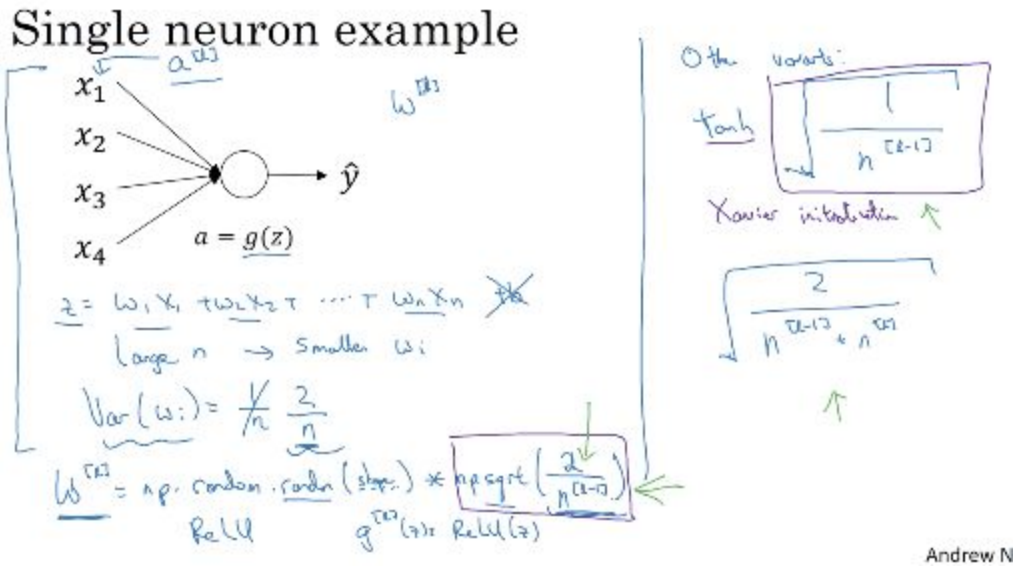


And

If each of the weight matrices $w^{[L]}$ is bigger than the identity matrix (say 1.5) we then eventually have $yhat = 1.5^{L-1} * X$ so if L is large because it comes from a deep neural network then yhat will be super large. Conversely, if we replace 1.5 with 0.5 then we have $yhat = 0.5^{L-1} * X$ and yhat ends up being very small. At the end of the day, the intuition is if we have $w^{[l]} > I$ then we have an exploding activation. If $w^{[l]} < I$ then we have the opposite case, the activations will decrease exponentially. The problem was shown with activations but the same case can be made for the derivatives or gradients. This is a problem because having small gradients means gradient descent will take small steps and will take a long time to converge.

Setting up the Optimization Problem: Weight Initialization for Deep Neural Networks

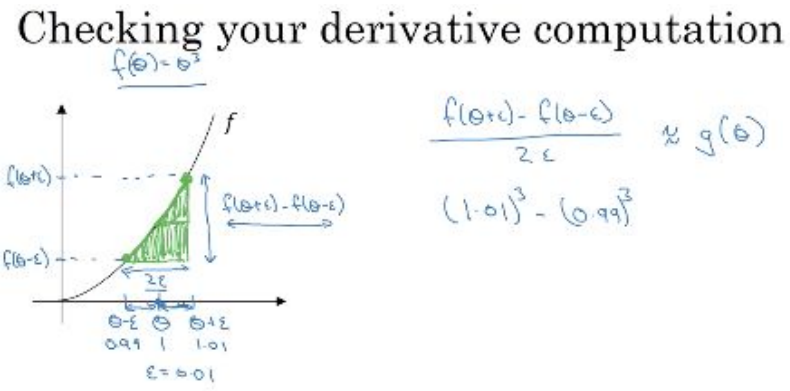
- A partial solution to vanishing/exploding gradients is the careful choice of random initialization for our neural network. Let's illustrate what we mean with a single neuron example. In the case below $z = w_1X_1 + w_2X_2 + ... + w_nX_n$. In order not to blow up or make z too small we want a small $w_{(i)}$ as n becomes larger. Why? Because z is the sum of all these terms and we want the terms to be small. One thing to do would be to set $variance(w_i) = 1/n$ where n = number of input features. In practice this translates to $w^{[l]} = np.random.randn(shape) * np.sqrt(1/n^{[l-1]})$. If we are using relu as our activation function setting the variance to 2/n works better.



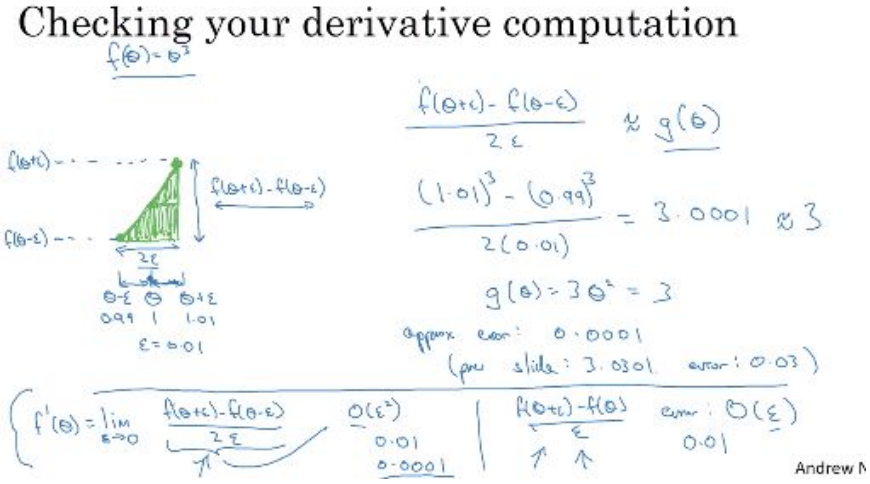
Other variants are: for tanh $variance = square\ root(1/n^{[l-1]})$. If we are using relu as an activation function we should stick with $2/n^{[l-1]}$ and if we are using tanh we should use $square\ root(1/n^{[l-1]})$.

Setting up the Optimization Problem: Numerical Approximation of Gradients

- Before getting into gradient checking let's check out how we can numerically approximate gradients. The chart below shows the function $f(\theta)$.

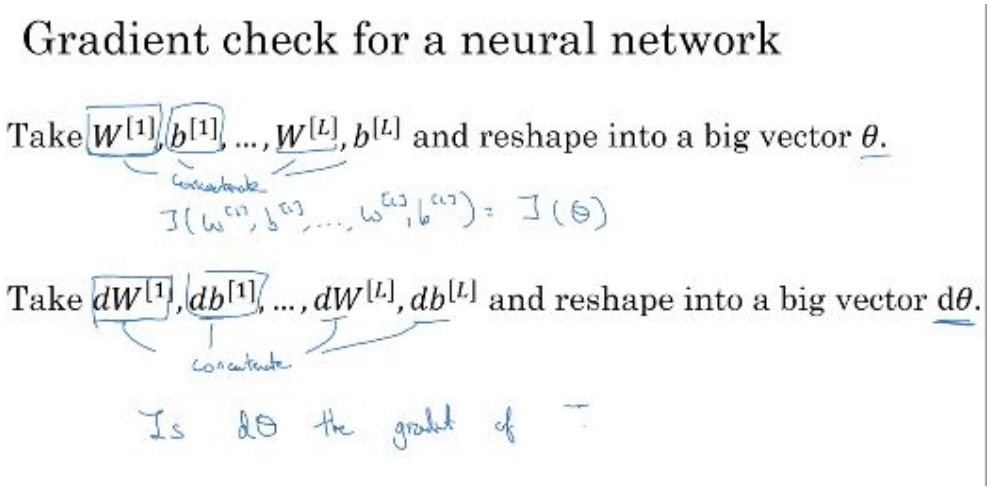


We know the derivative is the height/width but what we are going to do here is that we not only have theta at one point but also $\theta + \varepsilon$ and $\theta - \varepsilon$. We then calculate the derivative using those values and we have $(f(\theta + \varepsilon) - f(\theta - \varepsilon))/2\varepsilon$. ε is a small value, in this case 0.01. The derivative is then $(1.01)^3 - (0.99)^3/2(0.01) = 3.0001 \approx 3$. We knew that $g(\theta) = 3$ so our approximation is very close now, the difference is 0.0001. This gives us confidence that $g(\theta)$ is a correct approximation of the derivative. This method is a bit slower but worth the extra time since it is more accurate.

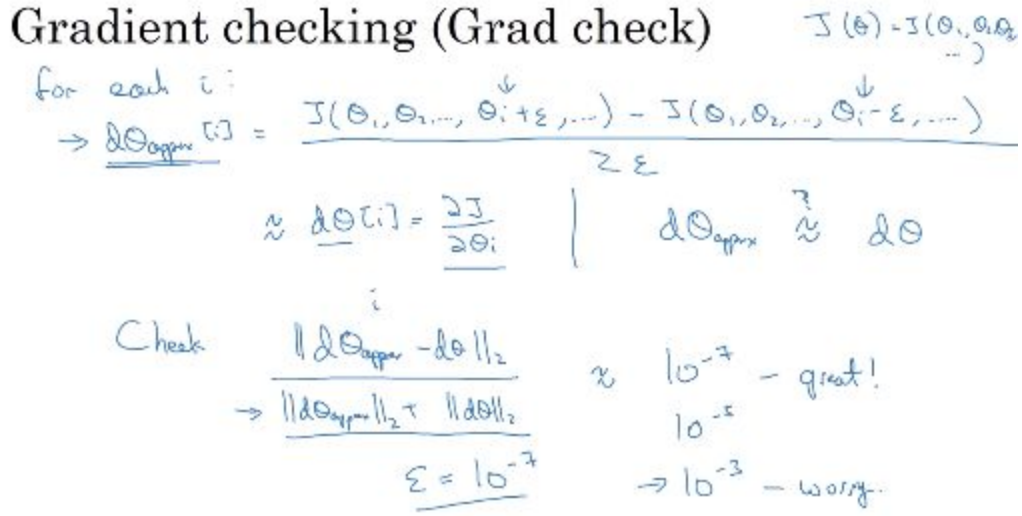


Setting up the Optimization Problem: Gradient Checking

- Gradient checking helps us to verify that our implementation of backprop is correct. We start by taking all the parameters $w^{[1]}, b^{[1]}, ..., w^{[L]}, b^{[L]}$ and make them a big vector θ and our cost function is now $J(\theta)$. We do the same to all the dw's and db's and we then have a big vector $d\theta$ .The question is now is if $d\theta$ is the gradient of $J(\theta)$.



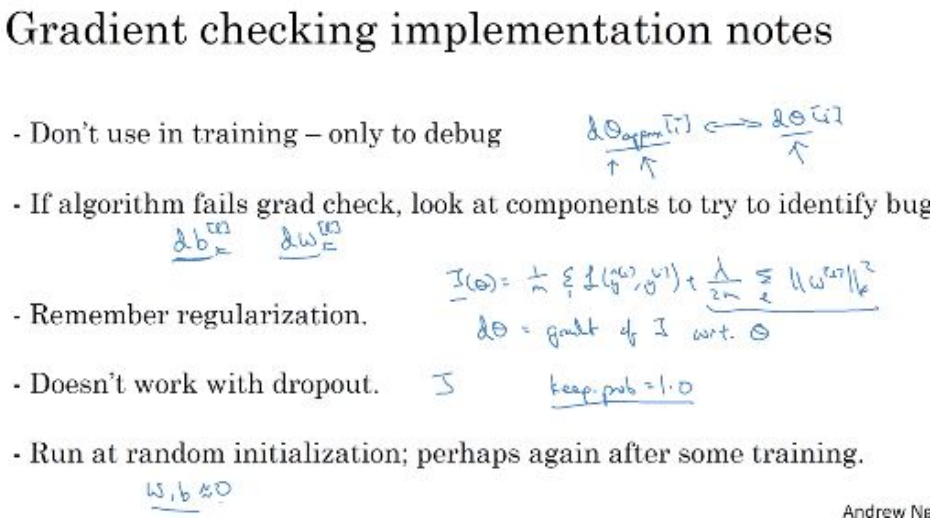
- We implement grad check using a for-loop where we have $for\ each\ i:\ d\theta_{approx}[i] = (J(\theta_1, \theta_2, ..., \theta_{i+\varepsilon}, ...) - J(\theta_1, \theta_2, ..., \theta_{i-\varepsilon}, ...)/2\varepsilon$ and this will give us the derivative $d\theta_{approx}[i]$. At the end we have two vectors $d\theta_{approx}[i]$ and $d\theta$ and the question is if they are very close or if $d\theta_{approx} \approx d\theta$. How do we know? What we do is we compute the euclidean distance between the 2 vectors which is $\|d\theta_{approx} - d\theta\|_2$ and it is the sum of the squares of the differences and then we take a square root. We then normalize by dividing by the lengths of the vectors so we end up with the following equation: $\|d\theta_{approx} - d\theta\|_2 / (\|d\theta_{approx}\|_2 + \|d\theta\|_2)$. In practice we use $e = 10^{-7}$. If our equation gives us 10^{-7} we are good. If it gives us 10^{-3} I would be worried and it could be an indication there is a bug and we need to go back to our theta vector components to check if there might be a derivative that is not computed correctly.



Summarizing, gradient checking helps us to make sure our gradient descent implementation is correct. If grad check is a big value then I need to go back and check if there is a bug and debug until I see that grad check has a small value. Then I am confident my gradient descent computation is correct.

Setting up the Optimization Problem: Gradient Checking Implementation Notes

- Some practical implementation notes for grad check are:
    - Don't use grad check in training, only for debugging because the grad check computation is too slow
    - If the algorithm fails grad check try to identify the parts/components that have the bug. It could be the problem is with the w parameters in one layer or the b parameters in another layer. We then try to hone in where the problem could be.
    - Remember regularization: if we are using regularization remember to include it in your calculations.
    - Grad check does not work with dropout. What Professor Ng recommends is to run grad check without dropout (so $keep.prob = 1.0$) and then turn on dropout.
    - Finally, run grad check at random initialization, then train the parameters a little while and then run grad check again.



Summary from first programming exercise: Initialization

- Different initialization methods lead to different results. In the exercise we tried 3 different initialization methods (zeros, large random values & He)
- Random initialization is used to break symmetry & to make sure different hidden units can learn different things
- Do not initialize to values that are too large
- He initialization works well for Relu activations

Summary from second programming exercise: Regularization

- L2 Regularization effect:
    - A regularization term is added to the cost function
    - The backprop function has extra term added to the gradients
    - Weights end up smaller (weight decay)
- Dropout is another regularization technique
    - We only use dropout during training, not during test time
    - We apply dropout to both forward and back propagation
    - During training time, we divide each dropout layer by keep_prob to keep the exact same value for the activations.
- Regularization
    - will help us to reduce overfitting.
    - will drive our weights to lower values.
    - L2 and dropout are 2 very effective regularization techniques.

Summary from third programming exercise: Gradient Checking

- Gradient checking verifies closeness between the gradients of back prop and the numerical approximation of the gradient (computed using forward propagation)
- Gradient checking is slow so we don't run it in every training iteration. We only run it once to make sure our code is correct, then we turn it off and use backprop