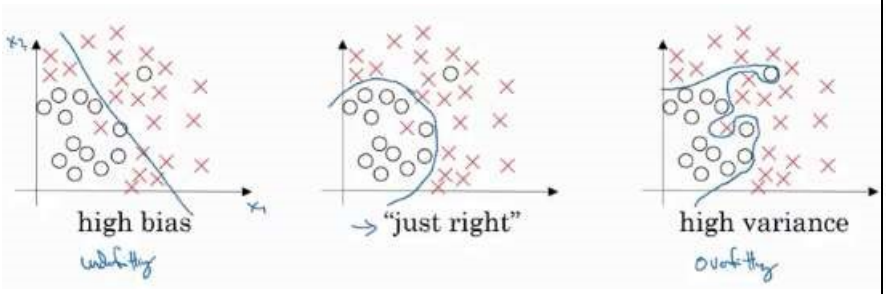
Course 2 Week1

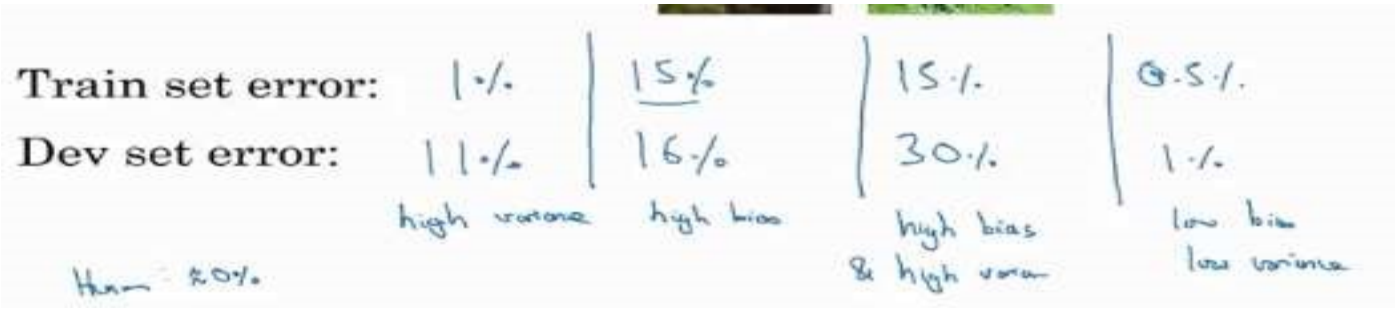
Datasets / Bias-Variance / Regularization: (recap)

Dataset is split to 3 parts : Training set(60%) / Development set (20%) / Test set (20%)
But in big data era dev and test sets are smaller than that as data now have become in range of millions of data so we don't need that much in dev and test sets so we increase the portion of training set.

Bias-Variance tradeoff:
High bias is called underfitting and High variance is called overfitting.

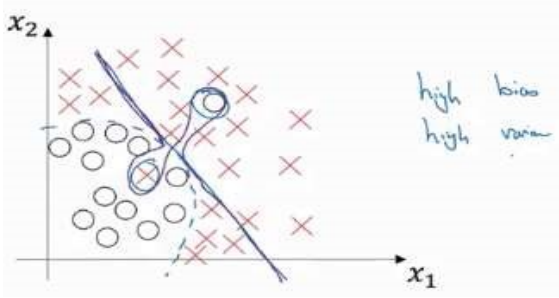


We say that we have high bias (underfitting) if the error in training set is much higher than the human-level performance error and we say we have high variance (overfitting) if the development set error is much higher than training error.
Assume we have human level performance error (AKA Bayes error) ≈ 0% . Therefore, ↓



Remember: Underfitting means your model is so bad to describe the data while overfitting means that your model is so good on the data that it can't generalize on unseen data. Both of them are not required to be found in your model.

Although there is a trade-off between bias and variance but sometimes it could happen that your model suffers from both as shown in this figure ➔

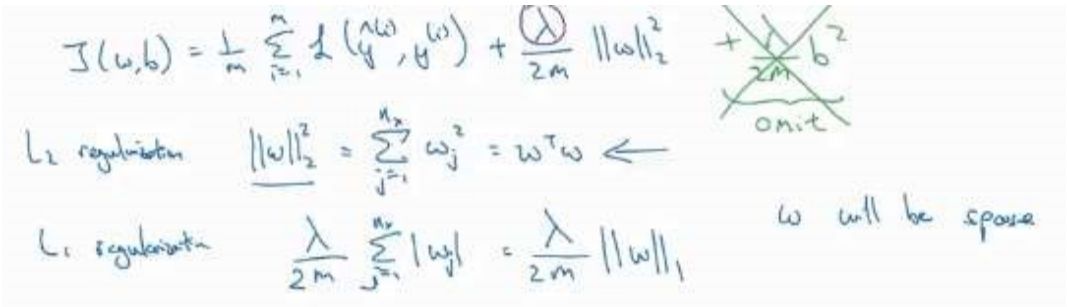


Basic recipe for learning: High bias?➔ • Bigger network High variance?➔ • More data
• Train longer • Regularization

Regularization:

It is used for reducing overfitting by adding regularization term either L1 or L2 (L2 is most common)

For logistic regression:



Note: We discard the regularization term for bias and just keep it for weights.

$\lambda \rightarrow$ regularization parameter.

For neural networks generally:

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

$\left\|w^{[l]}\right\|_F^2 = \sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}(w_{ij}^{[l]})^2$ ➔ This is called Frobenius norm (not L2 norm) noting that $w^{[l]}$ is a matrix of dimension $n^{[l-1]} \times n^{[l]}$.

Regularization is also in the update equation not only the cost function:

$$w^{[l]} := w^{[l]} - \alpha\left(dw^{[l]} + \frac{\lambda}{m}w^{[l]}\right) = w^{[l]} - \alpha\frac{\lambda}{m}w^{[l]} - \alpha\, dw^{[l]}$$

Remember:

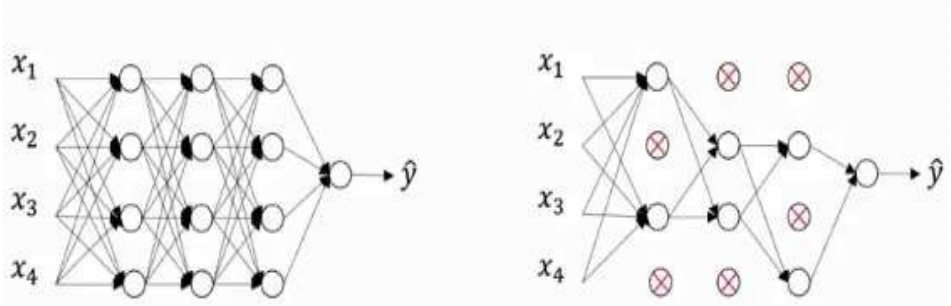
1-$dw^{[l]} \equiv \frac{\partial J}{\partial w^{[l]}}$ (Always check notations)

2-By increasing λ, you are decreasing weights (≈0) so your network becomes simpler and closer to linear network so increase the bias and decrease the variance.

Dropout regularization:

Dropout is a famous regularization technique used with neural networks by eliminating some neurons from each layer and keep the others. Removing nodes means by default removing all ingoing and outgoing links from that node so you end up with much smaller diminished simpler network less likely to overfit. Nodes to be eliminated is changed with each example after computing the backpropagation.

Removing some nodes helps also in making the kept nodes to learn more especially those who weren't participating enough in the learning process.



Keep probability (p) : It is the parameter that decides the percentage of neurons to be kept so if keep probability of layer 2 of network is 0.8 and this layer has 10 neurons then 8 of these neurons will be kept and only 2 will be eliminated from the layer.

There are different techniques to apply dropout as Inverted dropout which is the most common technique in dropout which is similar to original dropout explained so it keeps some weights and sets others to zero. The one difference is that, during the training of a neural network, inverted dropout scales the activations by the inverse of the keep probability q (q=1/p) . This makes the testing faster.

Note: For layers having more neurons so weight matrix is bigger so by default more likely to overfit so we give those layers with high number of neurons a lower keep probability to reduce overfitting caused by those layers.

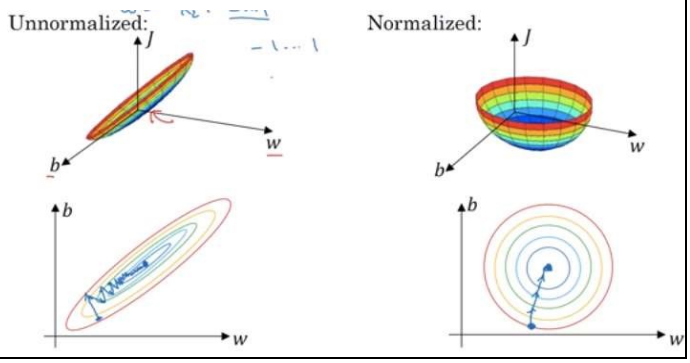
One of the drawbacks of dropout that it makes the cost function J no longer well-defined as each iteration we have removed bunch of neurons so J can't be drawn as a graph against the iterations and see something useful to check. The solution is to firstly not applying any dropout (keep probability =1) then after checking that J is monotonically decreasing (i.e. your algorithm is learning well) so you retrain again using dropout.

Other techniques of regularization:

- Data augmentation can be used by applying flipping, inverting, rotating, scaling, translating or changing lighting condition (This topic is discussed and figures shown in skewed classes part in machine learning part)
- Early stopping means to stop learning process at a specific iteration (epoch) where the point we stop is the place where training error and validation error start to move away as shown in the figure ➔

Early stopping has a drawback as it stops early to prevent overfitting, we are not getting the best optimized cost function as we stop gradient descent early so it is a trade-off between optimizing your cost function using gradient descent or any other optimization algorithm and preventing overfitting. This trade-off is also known as orthogonalization which will be discussed in details later.

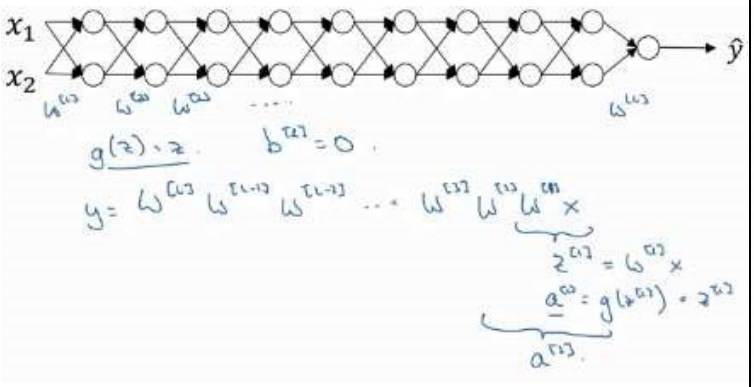
Remember: Don't forget to normalize or standarize your input features (normalize training sets) to make your data are of similar scales.



Vanishing/Exploding gradient descent:

These problems occur with very deep neural networks. This means that when training deep network, the derivatives or the slopes becomes either so big or so small (exponentially small).

To illustrate this, assume having a network as shown with a lot of hidden layer (deep network) and for simplicity assume we have linear activation function (g(z)=z) and bias =0. So the output y now equals the multiplications of all weights by the input x.



If we assume that each w is initialized as $\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$ except the last layer as it has different dimensions but now output y will be a value related to the multiplications of w or in other words y will be in terms of 1.5^L so it is exploding with more added hidden layers L. On the other hand, assuming w=$\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$ so y will be 0.5^L so it is vanishing with increasing number of hidden layers L.

A solution that helps a lot in solving these problems is carefully choosing the initialization of the weights. We have discussed this topic before and said to normalize our outputs by dividing weights by $\sqrt{n_i}$ where n_i is the number of neurons in the previous layer (inputs to this neuron) or use Xavier initialization discussed before.

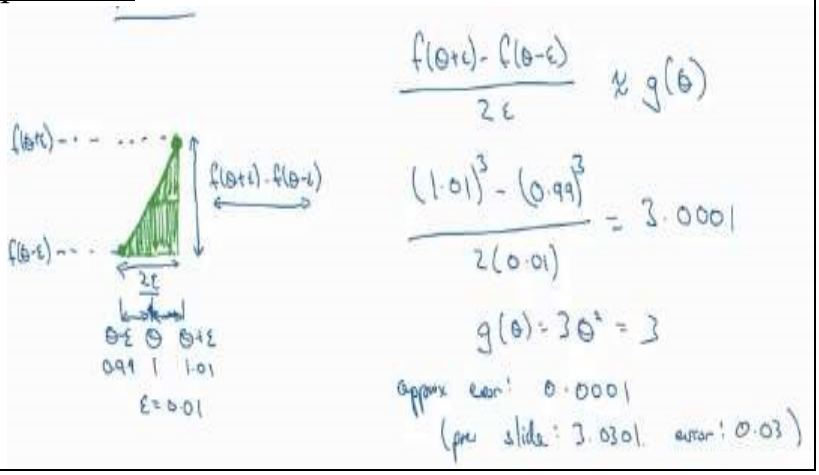
Note: It is found that if activation function is ReLU, it's better to use weights multiplied by $\sqrt{\frac{2}{n_i}}$ instead of $\sqrt{\frac{1}{n_i}}$.

Gradient checking:

Before getting into gradient checking, we want to talk about something.

Numerical approximation of derivative computation:

Derivative is known as discussed before by taking height over width of the triangle drawn between θ and θ+ε where ε is small number but by trying we discovered that by taking a bigger triangle between θ-ε and θ+ε the approximation is better numerically and gives better results. (without getting into calculus details)



Then ➔ $f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon}$ is better than $f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon)-f(\theta)}{\varepsilon}$

This new definition of derivative will be used in gradient checking

Note: usually use $\varepsilon=10^{-7}$

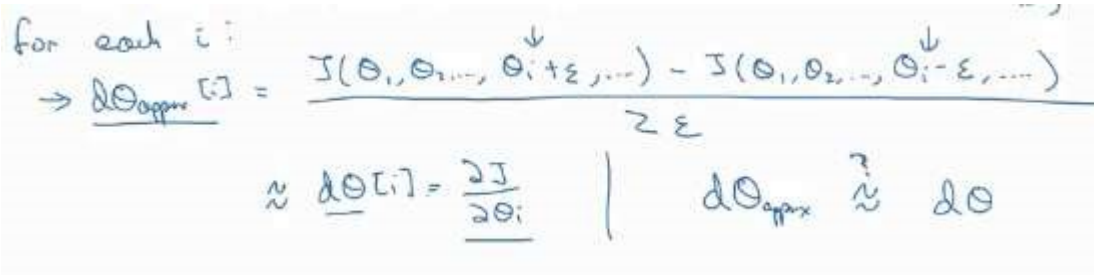
Gradient checking is a debugging method to verify that your backpropagation computation is correct.

Gradient checking algorithm:

We have W's and b's ($W^{[1]}$,$b^{[1]}$,… , $W^{[L]}$,$b^{[L]}$) so we gonna reshape them into big vector called θ such that each matrix w is reshaped into a vector then concatenate them all (W's and b's) to a vector called θ. Now instead of having J(W,b) we have J(θ).

Similarly with $dW^{[1]}$,$db^{[1]}$,… we going to reshape them and concatenate to form big vector called dθ.

Now ask yourself is dθ is the gradient or the slope of J(θ)?



We will apply the derivative rule to each of J(θ) noting that J(θ) = $J(\theta_1,\theta_2,\ldots\theta_L)$ so if $d\theta_{approx}$ that is computed from the derivative rule is close to that dθ we have made before so our backpropagation was done right. But the question is how to know that $d\theta_{approx}$ is close to dθ?

The Check is ➔ $\frac{\|d\theta_{approx}-d\theta\|}{\|d\theta_{approx}\|+\|d\theta\|}$ so if the result in range of 10^{-6} or 10^{-7} so it is OK if it is bigger then more likely there is a bug somewhere in computing gradients.

Notes:

1-Include regularization term in your check.

2-Grad check doesn't work with dropout.

3-Don't use it across your training but just once for debugging.