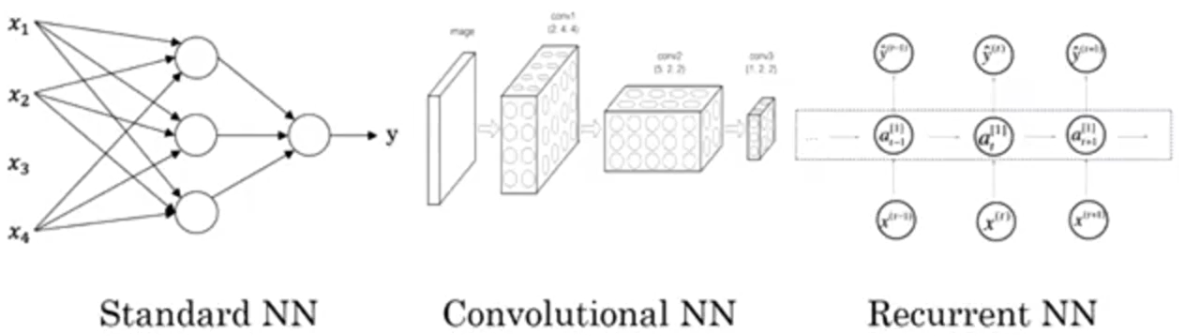
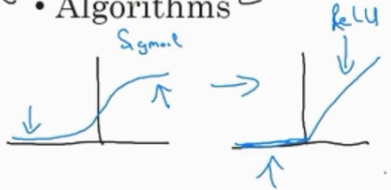
**Deep Learning Specialization**

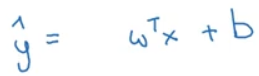
**Neural Networks and Deep Learning**

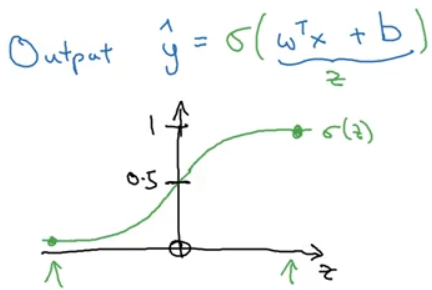
* **Supervised Learning**
  + Both input and output are given to the model for training.
  + Different NN are used for different application
    - For real estate(price of a house) and online advertising standard NN is used
    - For image applications like photo tagging CNN is used
    - For speech recognition(audio to text), audio is a one dimensional temporal sequence. For sequence data RNN is often used
    - For language translation, e.g. English to Chinese. The alphabets/words come one after the other, so languages are also sequence data. So RNN is used
    - For autonomous driving, for image data CNN is used and for radar data custom or hybrid NN is used.

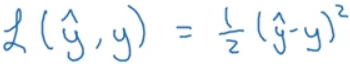


* + Switching the activation function from Sigmoid to ReLU has made the gradient decent algorithm work much faster as the Sigmoid approaches a slope of 0 on both sides of the function. With slope approaching 0, the parameters of the gradient decent algorithm changes by a very small amount every iteration thus making the learning much slower.

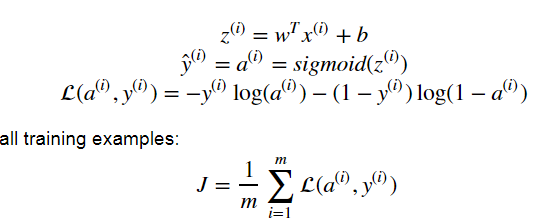


* **Logistic Regression as a NN**
  + , Just using the weights and bias will make it a linear function and the model will be a linear regression model. So the sigmoid function is used to make it a logistic regression model

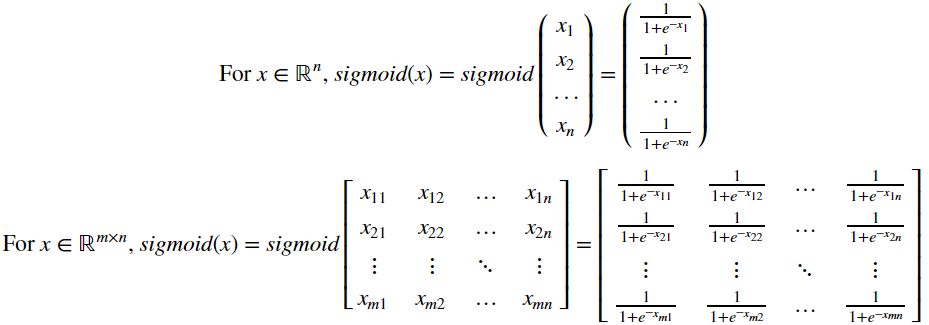


* + A Loss function such as squared error, , can be used but the optimization problem using this loss function can become non-convex(multiple local optima ) and gradient decent may not find the global optimum. So, for logistic regression problem, the loss function,  , is chosen which gives an optimization problem which is convex (). As this is a logistic regression problem, y can take either 1 or 0. Check what the loss function has to do when y is 1 or 0. The loss has to be minimum, so the optimization problem will push y\_hat in the direction such that loss becomes minimum. The loss function is defined on a single training data. For the entire training set, a Cost function is defined  as average of the loss function applied to each training data.

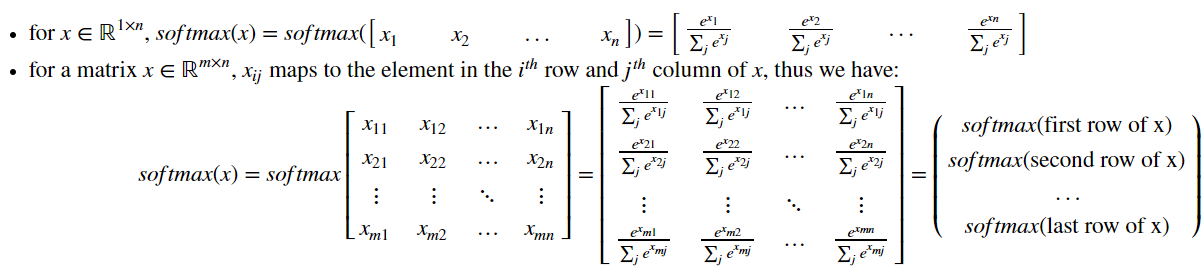
Summary:



* **Python and Vectorization**
  + Sigmoid function:

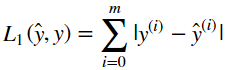


* + Softmax function:



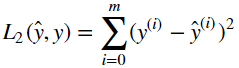
* + **L1** loss functions

Also known as **Least Absolute Deviations(LAD)/Least Absolute Errors(LAE)**

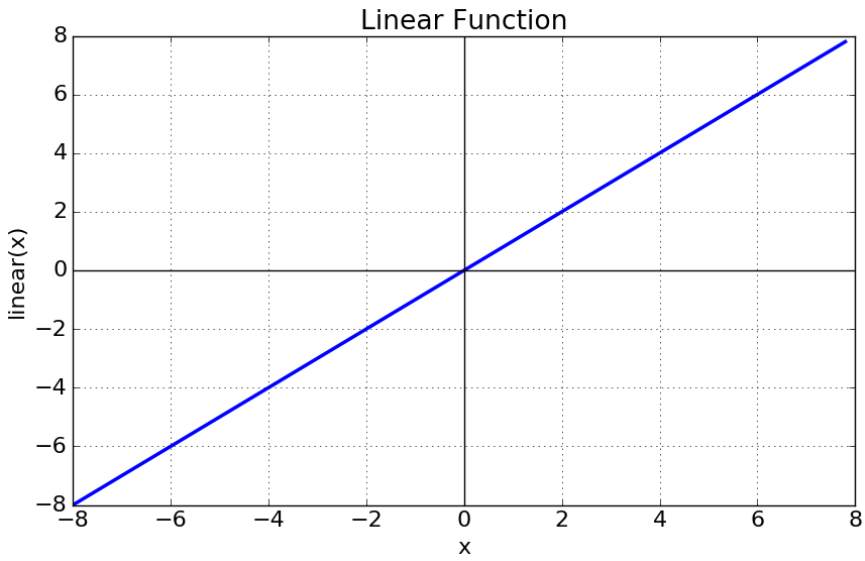


* + **L2** loss function

Also known as **Least Square Errors(LS)**

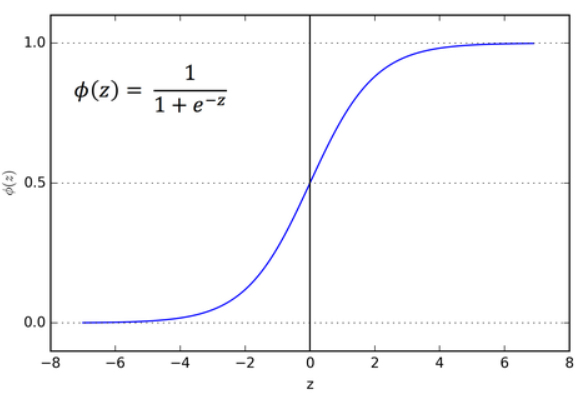


* **Activation layers**
  + **Linear**



Should be used only for very simple, linear input data.

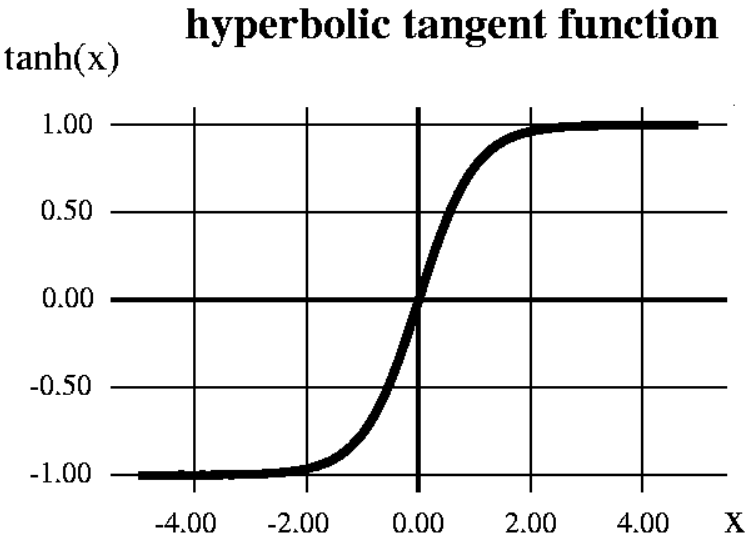
* + **Sigmoid**



Never use it for the hidden layers except for the output layer if we are doing binary classification. It can be used for models where we have to predict the probability of an output(as it lies within the range of 0 and 1).

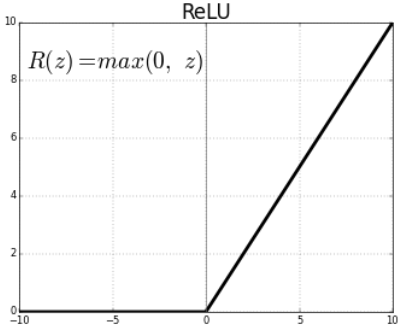
But both Sigmoid and Tanh suffer from vanishing gradient problem, i.e. the gradient reaches zero faster and makes learning very slow(the parameters gets updated by a extremly small number).

* + **Tanh**



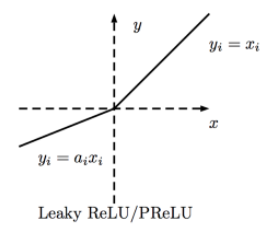
Its range is -1 to 1, so the mean of the output is closer to zero which implies it centers the data better for the next layer and hence it is preferred over Sigmoid.

* + **ReLU**



The gradient will be 1 most of the times which avoids the vanishing gradient problem. Its limitation is that it should only be used with the hidden layers. Another problem with ReLU, due to its 0 value for negative numbers, is dead neurons.

* + **Leaky ReLU**

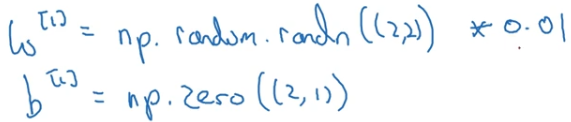


This answers the dead neurons problem.

If we simply use linear activation function, then the output will just be a linear combination of the input no matter how many hidden neurons/layers are used. Most of the time the input-output relation is very complex and non-linear activation functions can mimic those complex functions/relation.

* **Random Initialization**

It is not a good idea to initialize the all the weights to zeros(it is fine to initialize the bias to zeros) as all the neurons in the hidden layer will have the same output and during back propagation, all the weights will be updated to the same value. This will be equivalent to having only one neuron in the hidden layer though there are many. It is usual practice to initialize the weights to random numbers and bias to zeros as below:

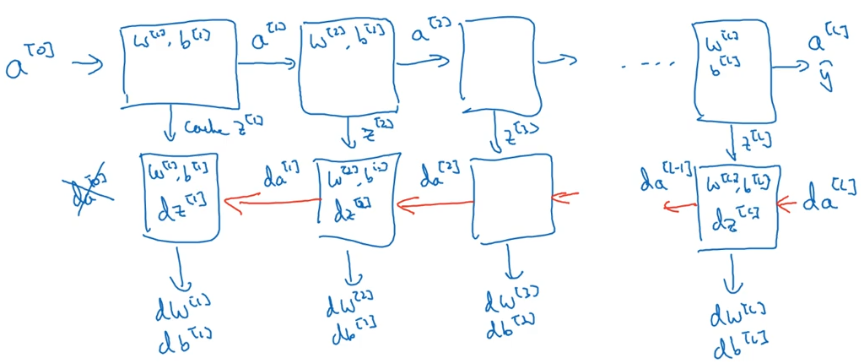


The weights are further multiplied with a small number and this number purely depends on the activation function used. For e.g., if the activation function is sigmoid or tanh then having bigger weights implies the output value will be big and will fall in the saturation zone of the activation

function where the gradient is zero and there will be no learning or slow learning.

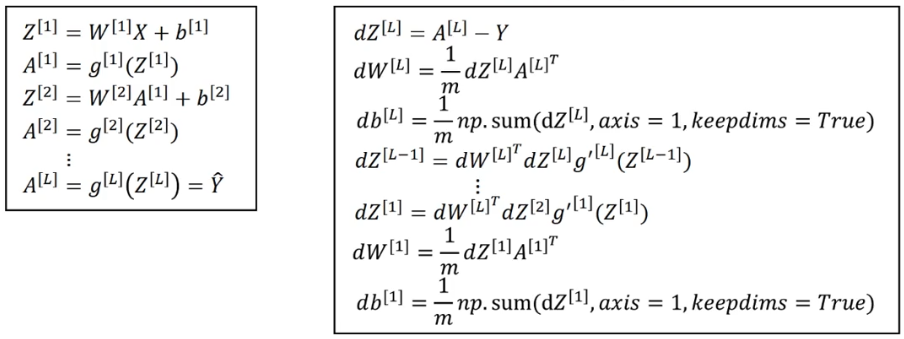
* **Deep Neural Network**

The forward and backward propagation of a neural network with L layers



The forward propagation takes in input as previous layer’s activation value and outputs current layer’s activation value. It also caches activation value, neuron’s output(Z), weights and bias which will be later used in backward propagation. The backward propagation takes in input as derivative of the previous layer’s activation and outputs derivative of the current layer’s activation. It also outputs the derivative of the respective layer’s weights and bias which will be used for updating the weights and bias.

Steps for forward and backward propagation(this is very specific to the loss function being used)



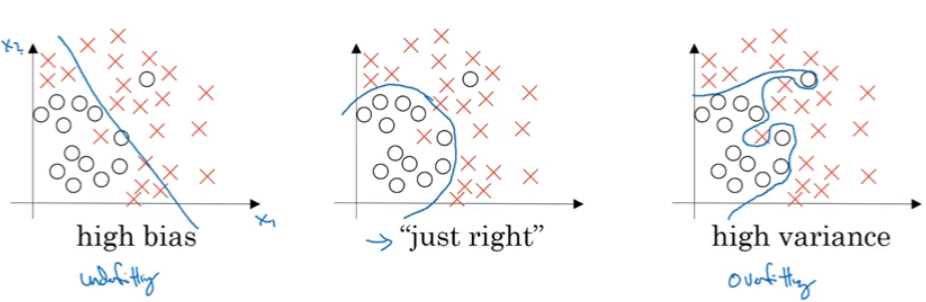
**Improving Deep Neural Networks: Hyper parameter tuning, Regularization and Optimization**

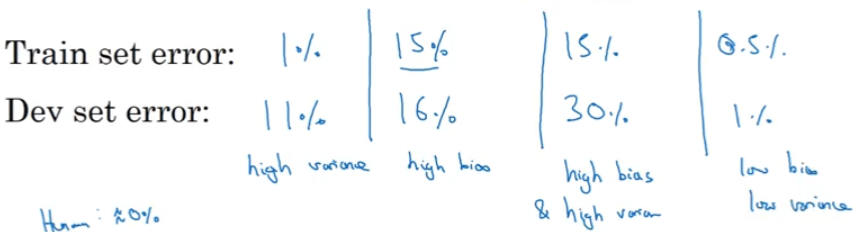
* **Bias-Variance**

Bias is checked by looking at the model’s performance on training set. If the model is not performing well on the training set, maybe because the model is too simple and not able to learn all the features of the dataset, then the model has high bias. The model is also said to be under-fitting.

Variance is checked by looking at the model’s performance on development set. If the model is not performing well on development set, maybe it learnt even the noise in the training set instead of just learning the general trend, then the model has high variance. The model is also said to be over-fitting.

If the model doesn’t perform well on both training and development set, then the model has high bias and high variance which is the worst model. If the model performs well on both training and development set, then the model has low bias and low variance which is the best model.





* **Regularization**

Regularization often helps in reduce high variance(over-fitting) in the model.

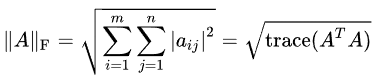
* + **Weight penalty**

One kind of regularization is to penalize the weights. The weight terms are added to the cost function. Higher weights implies higher cost, so the model tries to make the weights small in order to reduce cost. If the weights tend to be small then the model(which was having high variance) tends to be simple and the model will not be able to learn the noise in the dataset thereby reducing overfitting. If the absolute value of the weights are used, then it is called L1 regularization and if the squared weight are used, then it is called L2 regularization. While the L2 regularization tends to drive all the weights to smaller values, L1 regularization tends to drive some weights to exactly zero(introducing sparsity in weights) and allowing some weights to be big. Sparsity helps in reducing the memory required to save the weights.

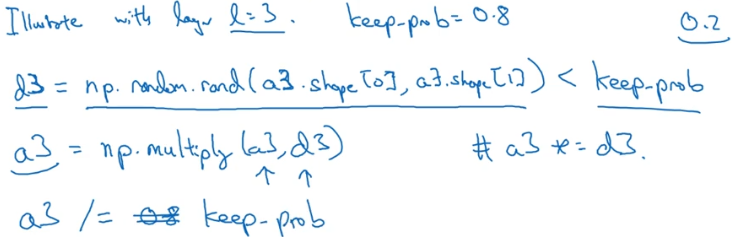
L1 : 

L2 : 

L2 norm for a matrix is called Frobenius norm and is the square root of sum of all the elements of the matrix.



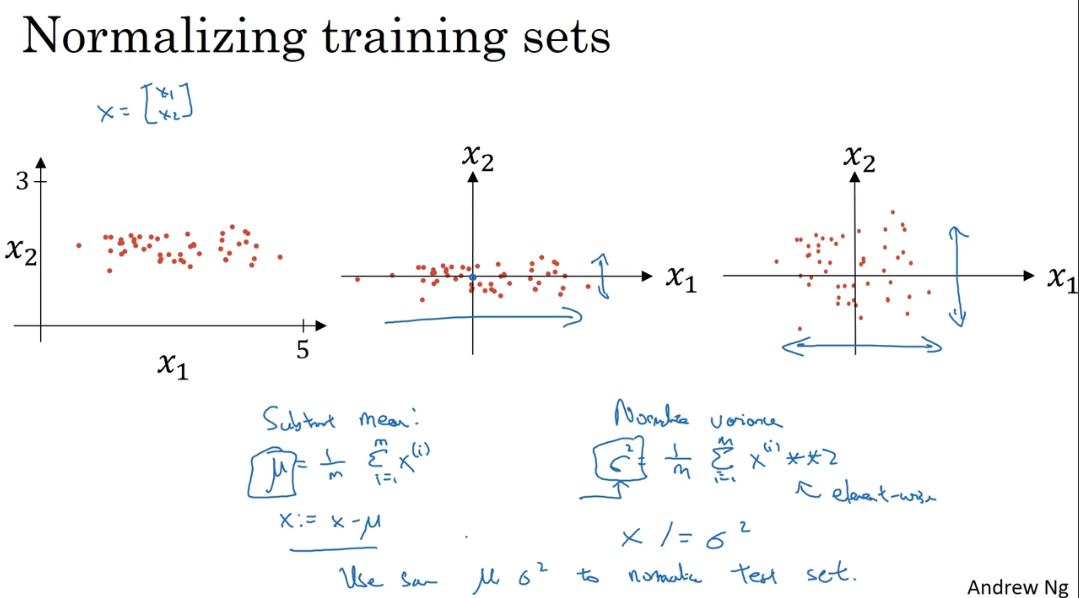
* + **Inverted dropout**



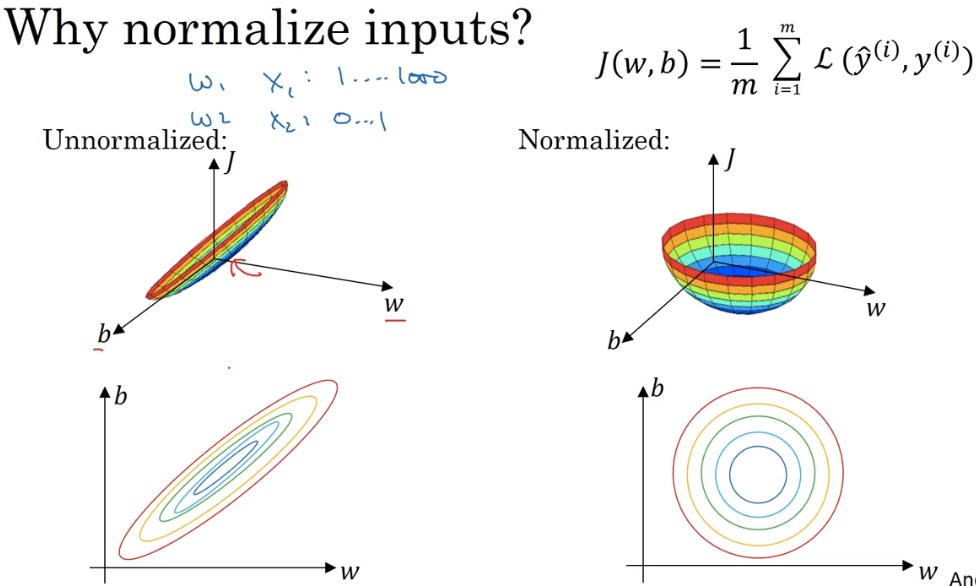
Above is how the concept of dropout works. A random matrix/vector, the size of the activation layer, is compared against the parameter keep\_prob. This resulting Boolean matrix/vector is multiplied with the activation layer which then makes around (1-keep\_prob)% (20% for above example) of the activation layer as zeros. Finally the activation layer is scaled by inverse of keep\_prob(hence the name Inverted dropout) to keep the expected value(mean) of the activation layer similar to what it was before dropout. Dropout is not used in the prediction step because it would lead to inconsistent output every time you run the prediction on same input.

By introducing dropout, the model cannot rely on any one feature as random neurons get dropped for every iteration. So the model will have to spread out the weights making the weights small. Again as explained in Weight Penalty section, smaller weights imply simpler models and hence reducing over-fitting.

* + Data augmentation
  + Early stopping
* **Normalizing inputs**



The above input is not close to zero and variance in x1 is larger when compared to variance in x2. Subtract the mean to zero center the data and then divide by variance so that the variance in all dimensions are similar.



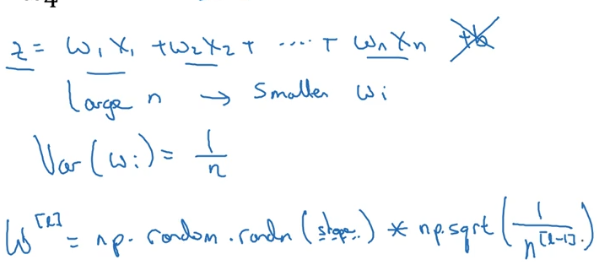
Above is the graph and contour map of the cost function for un-normalized and normalized inputs. The normalized input leads to a nice symmetric contour for the cost function. To run the gradient descent on the cost function of un-normalized input, we might have to use a very small learning rate as it takes a lot of steps, oscillating a lot, to finally reach the minimum. If we normalize the input, then irrespective of where it starts the gradient descent can pretty much go straight to the minimum.

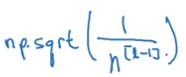
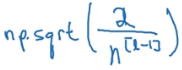


The cost function will be more symmetric and easier to optimize when the features are on similar scale.

* **Weight initialization for Deep networks to avoid vanishing/exploding gradients**

To avoid vanishing/exploding gradient issue, different initialization methods are used based on the activation function being used. They all try to make the variance of the weights equal to (1/number-of-weights)

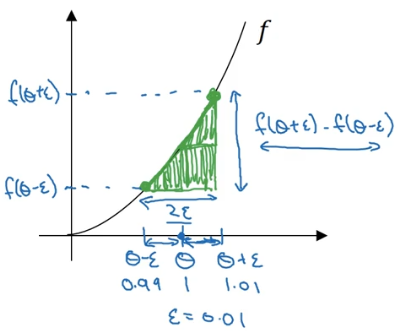


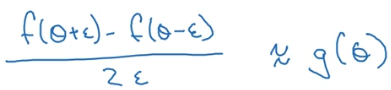
The above multiplication term, , is used if the activation function is tanh. For ReLU,  is used.

* **Gradient Checking**

Once the parameters(W1, b1, W2, b2, …, WL, bL) and their gradients(dW1, db1, dW2, db2, …, dWL, dbL) are available, it is good practice to check the credibility of the gradients.

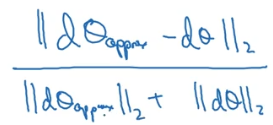
The gradient of the parameter is calculated using the numerical approximation(two sided difference) of gradient:



 g(θ) is the derivative of f(θ).



This numerical approximation of the gradient should be similar to the gradient calculated by the model:

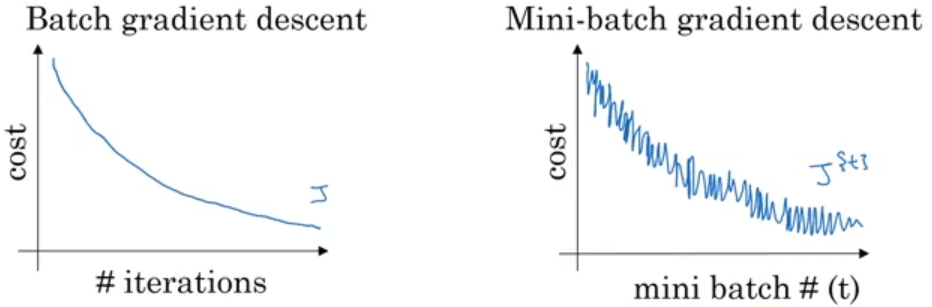
 This ratio should be very small number.

Don’t forget to include the regularization term if it is used in cost computation.

Gradient check doesn’t work with dropout.

* **Algorithm optimizations**
  + **Mini-batch gradient descent**

A large training set is divided into smaller training sets called mini-batches. The gradient descent algorithm is run on each of these mini-batches one after the other. Once all the mini-batches are processed then we say the algorithm has gone through one epoch.

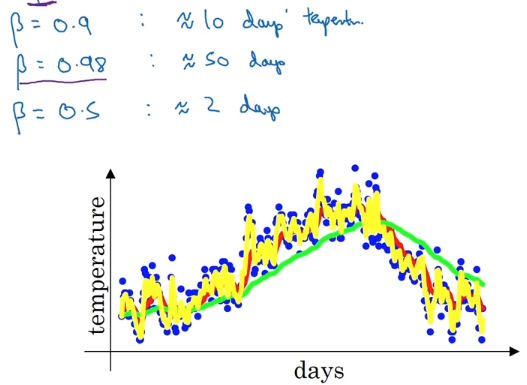
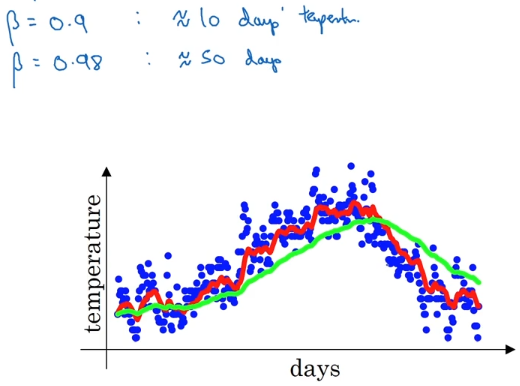


When the model is good, we expect the cost to decrease every iteration for a batch gradient descent. The number of sample points on the left image will be equal to number of iterations. When we use the mini-batch gradient descent, the cost will not necessarily decrease for every batch as one batch could be very good for the model(thus lower cost) and the next batch could be bad(thus higher cost). This leads to the oscillation as shown on the right image. But overall, the model should show a general trend of decreasing cost. The number of samples on the right image for one epoch will be equal to number of mini-batches.

* + **Gradient descent with momentum**
    - **Exponentially weighted moving averages(EWMA)**

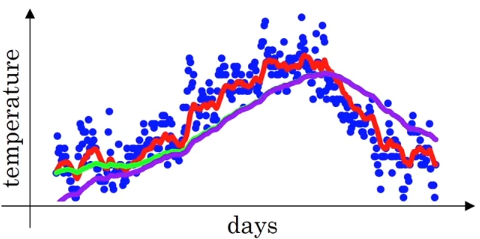


Instead of taking the sample values as it is, which might be noisy, it is better to take average value of the last **n** samples. **n**  is approximately equal to 1/(1-β). If β=0.9, then the averaging will be over last 10 samples. For bigger value of β, the result will be a much smoother curve as it averages over a large number of previous samples but this might not capture the general trend. For smaller values of β, the result will capture the noise in the data.



The formula 1/(1-β) is based on the fact that with . If , then 0.9 has to be raised to 10 to get a value approximately equal to .

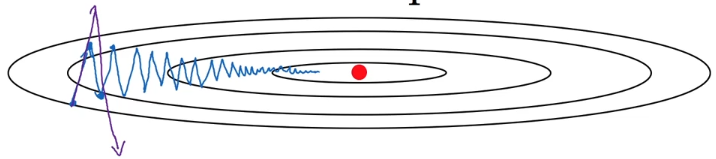
* + - **Bias correction in EWMA**



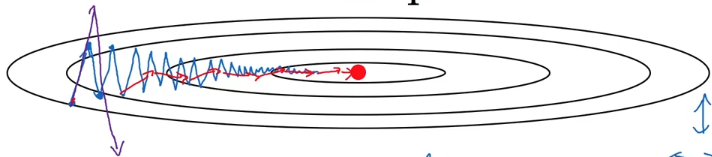
Purple curve is the EWMA and green curve is the EWMA with bias correction. The purple curves results due to the fact that the previous EWA term is assumed 0 for the first time. The bias correction tells us to divide the average term by , 

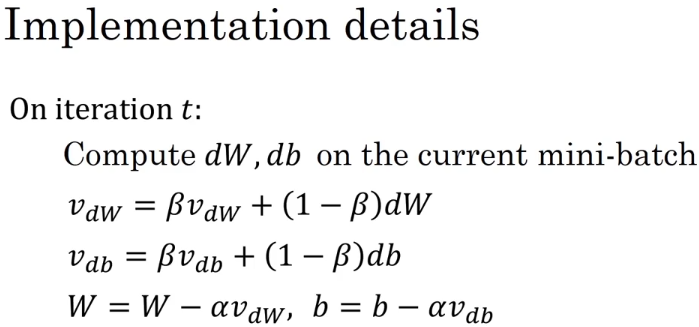
In this algorithm, the EWMA of the gradients is calculated and this averaged gradient is used to update the weights and bias. This algorithm almost always works faster when compared to the standard gradient descent algorithm.

Below is how a standard gradient descent algorithm would work on the contour of the cost function:

There is lot of oscillation which slows down the algorithm and also prevents us from using larger learning rate as larger learning rate would mean bigger steps which might lead to overshooting.

Below is how a gradient descent with momentum would work:

 The algorithm can reach the minimum much faster as there are less oscillation and also increases the chance of using higher learning rates.



* + **RMSprop**