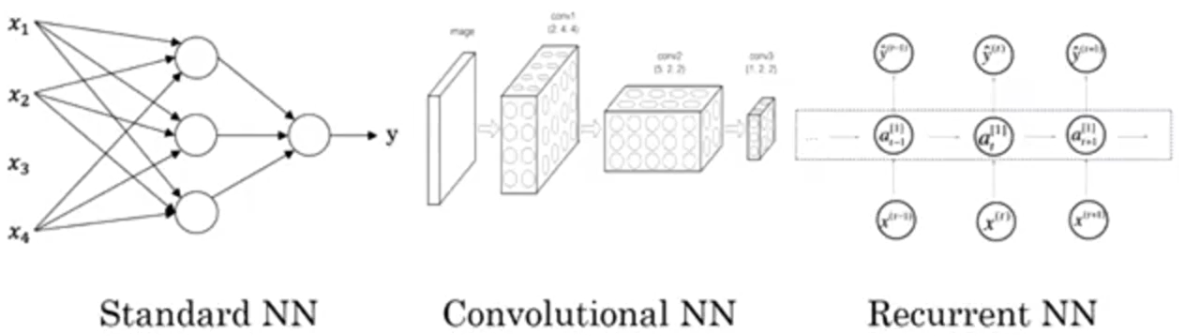
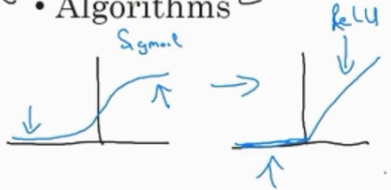
**Deep Learning Specialization**

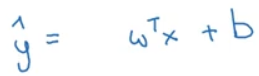
**Neural Networks and Deep Learning**

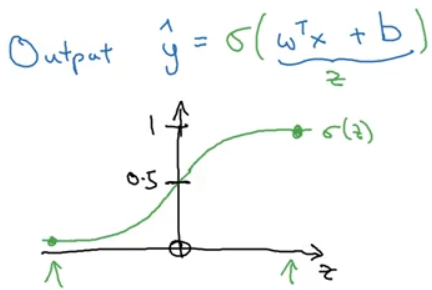
* **Supervised Learning**
  + Both input and output are given to the model for training.
  + Different NN are used for different application
    - For real estate(price of a house) and online advertising standard NN is used
    - For image applications like photo tagging CNN is used
    - For speech 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.

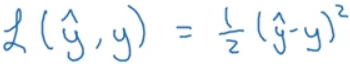


* + Switching the activation function from Sigmoid to ReLU has made the gradient decent algorithm work much faster as the Sigmoid approaches a slope of 0 on both sides of the function. With slope approaching 0, the parameters of the gradient decent algorithm changes by a very small amount every iteration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

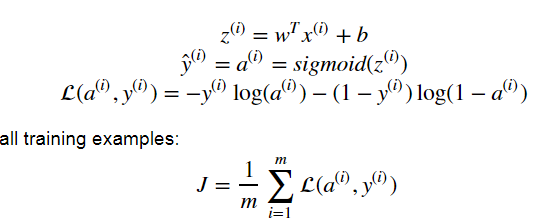


* **Logistic Regression as a NN**
  + , Just using the weights and bias will make it a linear function and the model will be a linear regression model. So the sigmoid function is used to make it a logistic regression model

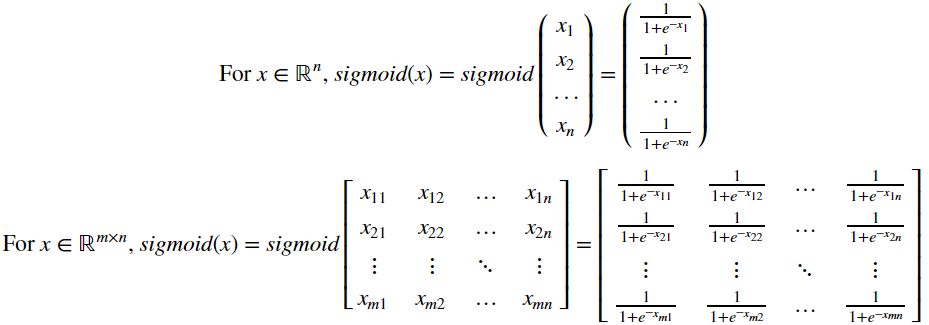


* + A Loss function such as squared error, , can be used but the optimization problem using this loss function can become non-convex(multiple local optima ) and gradient decent may not find the global optimum. So, for logistic regression problem, the loss function,  , is chosen which gives an optimization 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:

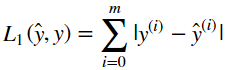


* + Softmax function:



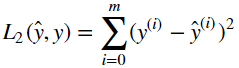
* + **L1** loss functions

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

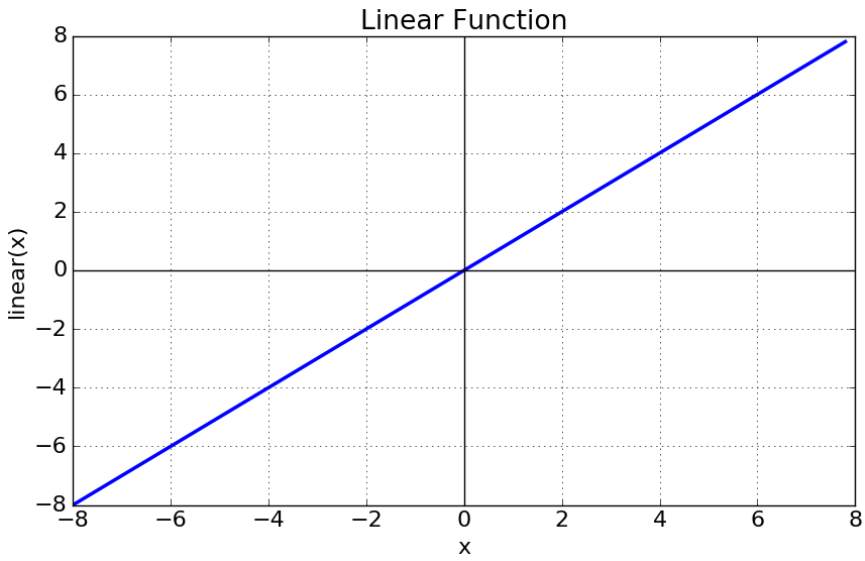


* + **L2** loss function

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

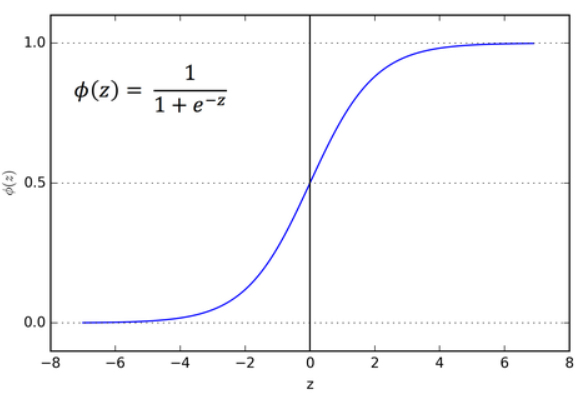


* **Activation layers**
  + **Linear**



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

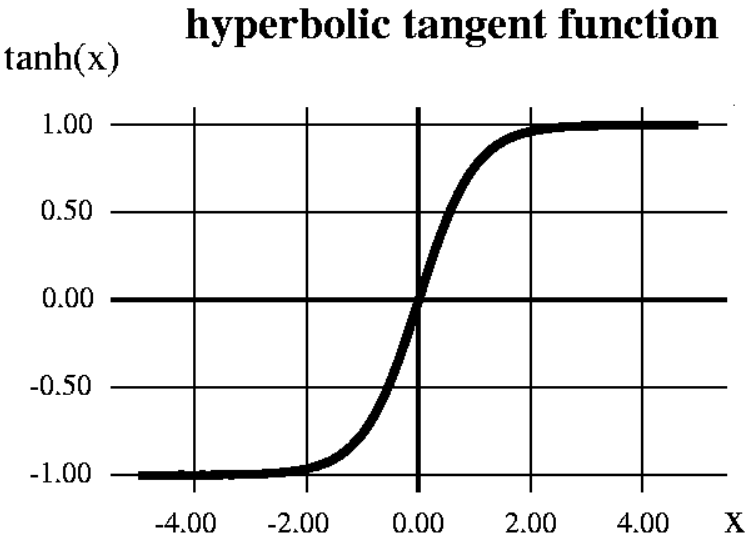
* + **Sigmoid**



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).

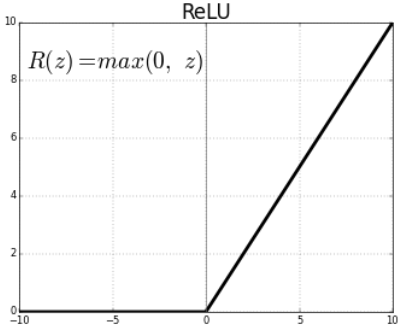
But both Sigmoid and Tanh suffer from vanishing gradient problem, i.e. the gradient reaches zero faster and makes learning very slow(the parameters gets updated by a extremly small number).

* + **Tanh**



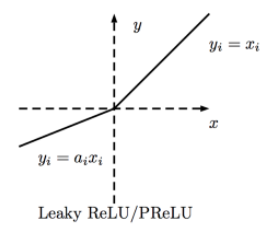
Its range is -1 to 1, so the mean of the output is closer to zero which implies it centers the data better for the next layer and hence it is preferred over Sigmoid.

* + **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**

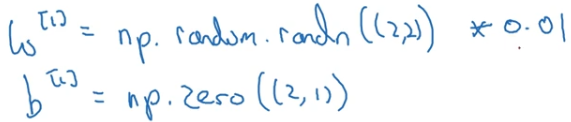


This answers the dead neurons problem.

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:

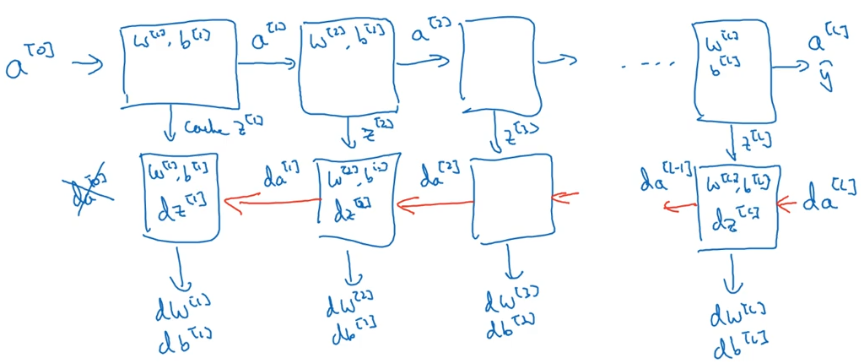


The weights are further multiplied with a small number and this number purely depends on the activation function used. For e.g., if the activation function is sigmoid or tanh then having bigger weights implies the output value will be big and will fall in the saturation zone of the activation

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

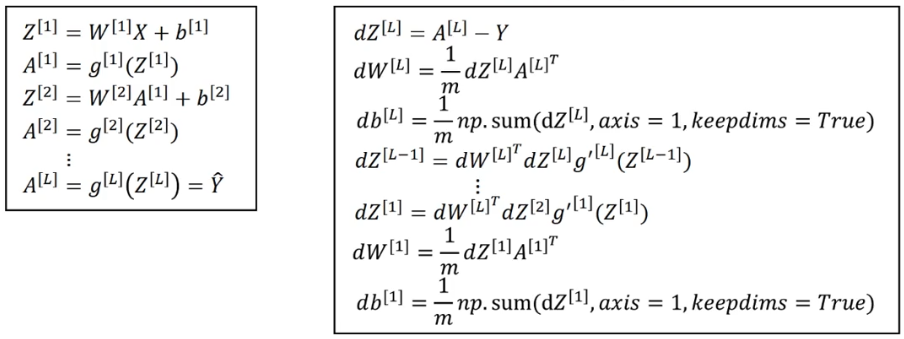
* **Deep Neural Network**

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)



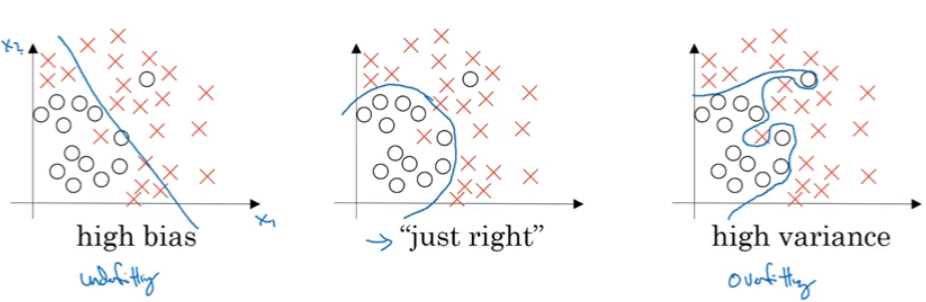
**Improving Deep Neural Networks: Hyper parameter tuning, Regularization and Optimization**

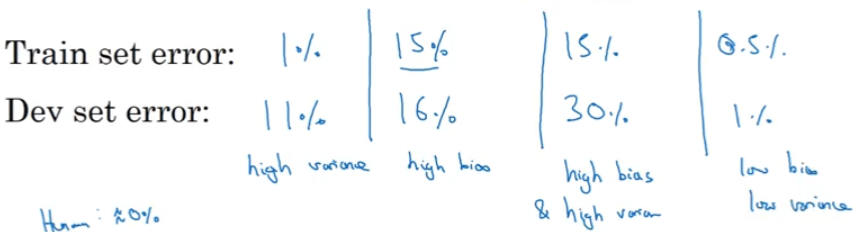
* **Bias-Variance**

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.

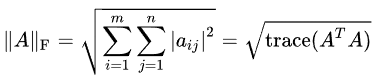
* + **Weight penalty**

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.

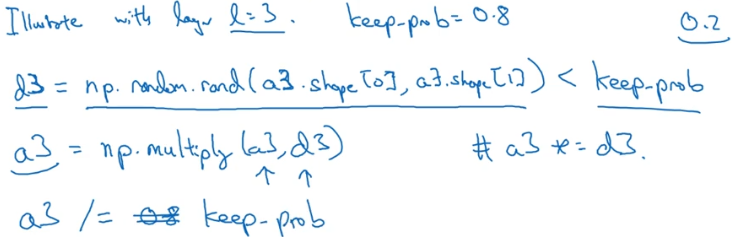
L1 : 

L2 : 

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



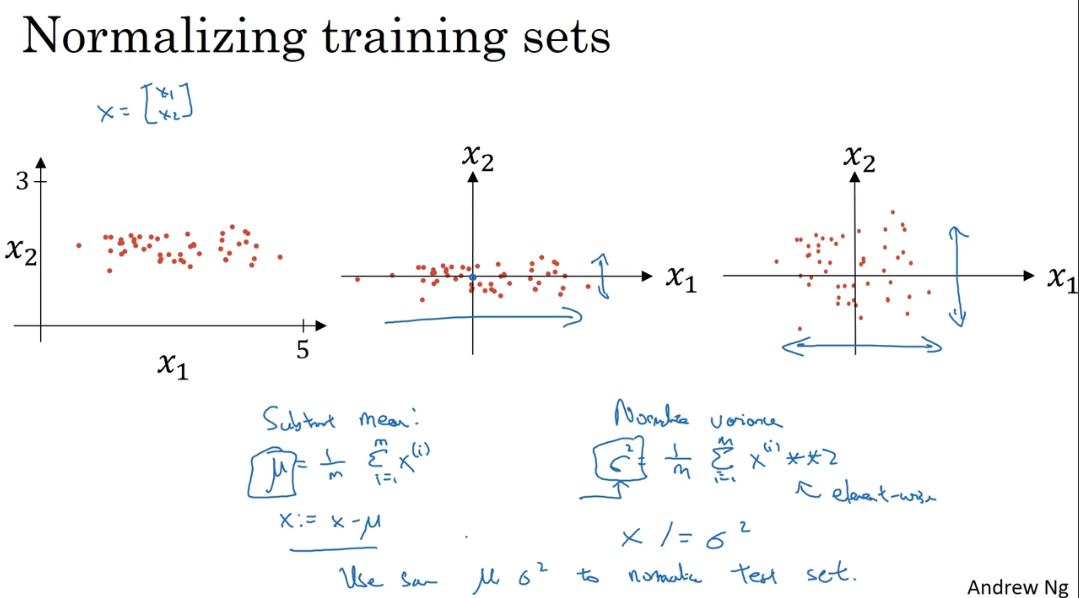
* + **Inverted dropout**



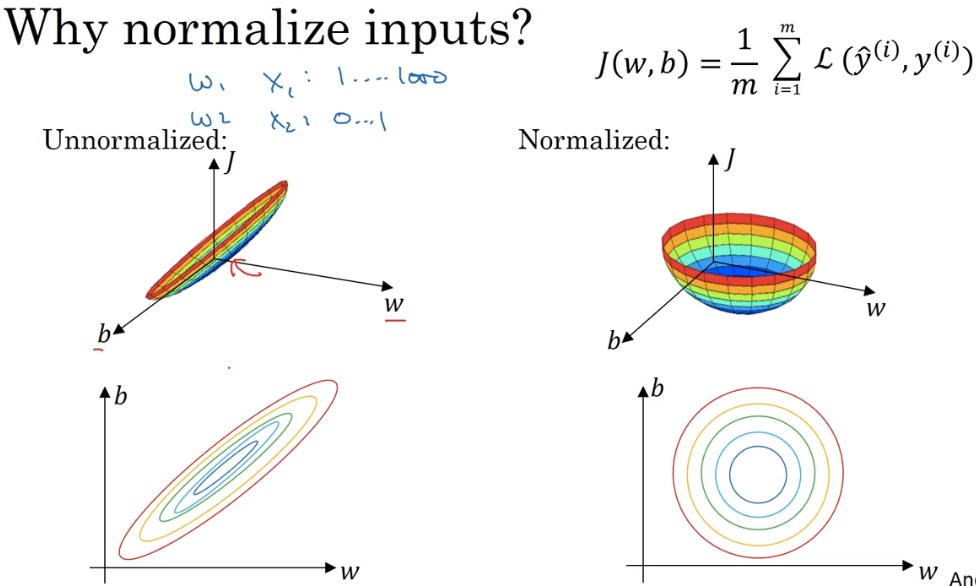
Above is how the concept of dropout works. A random matrix/vector, the size of the activation layer, is compared against the parameter keep\_prob. This resulting Boolean matrix/vector is multiplied with the activation layer which then makes around (1-keep\_prob)% (20% for above example) of the activation 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.



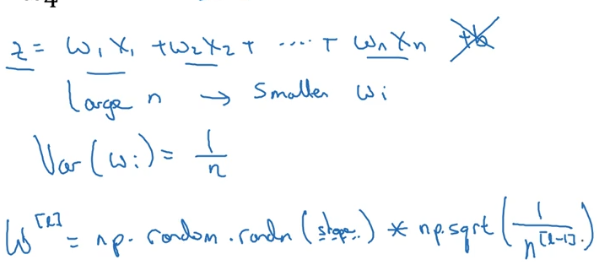
Above is the graph and contour map of the cost function for un-normalized and normalized inputs. The normalized input leads to a nice symmetric contour for the cost function. To run the gradient descent on the cost function of un-normalized input, we might have to use a very small learning rate as it 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.

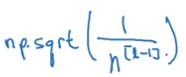
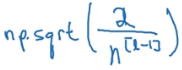


The cost function will be more symmetric and easier to optimize when the features are on similar scale.

* **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)

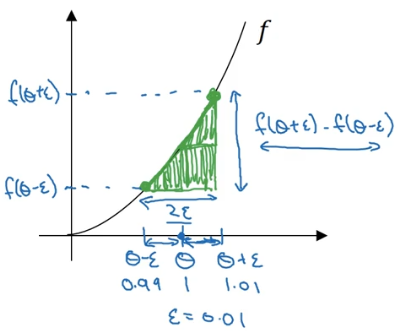


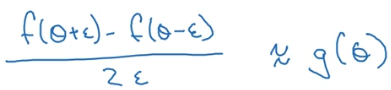
The above multiplication term, , is used if the activation function is tanh. For ReLU,  is used.

* **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.

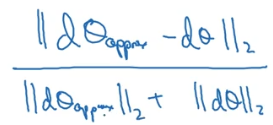
The gradient of the parameter is calculated using the numerical approximation(two sided difference) of gradient:



 g(θ) is the derivative of f(θ).



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

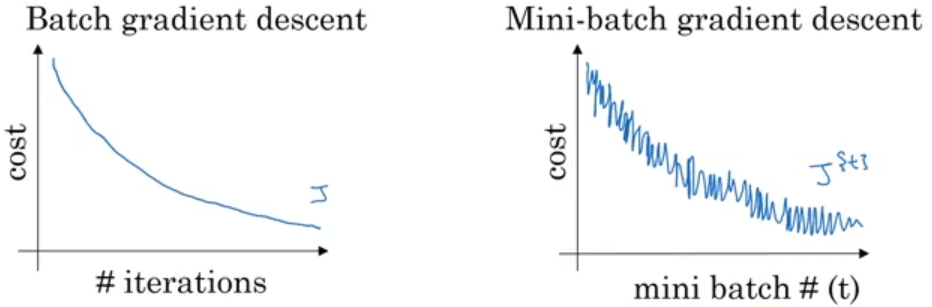
 This ratio should be very small number.

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.

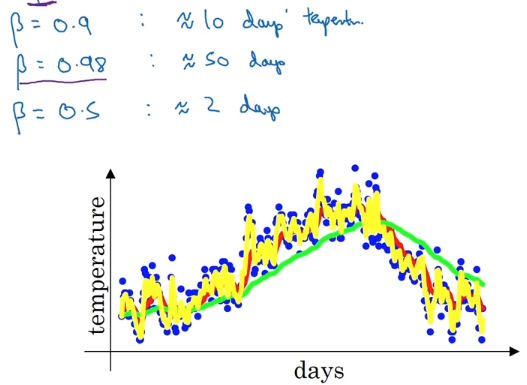
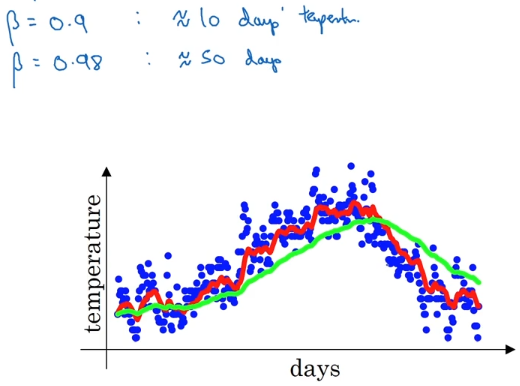


When the model is good, we expect the cost to decrease every iteration for a batch gradient descent. The number of sample points on the left image will be equal to number of iterations. When we use the mini-batch gradient descent, the cost will not necessarily decrease for every batch as one batch could 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. If the mini-batch size is equal to total number of training data then it is just called Batch Gradient Descent. If the mini-batch size is 1 then it is called Stochastic Gradient Descent.

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

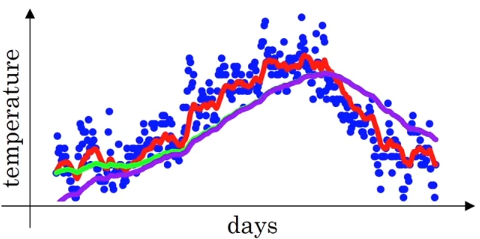


Instead of taking the sample values as it is, which might be noisy, it is better to take average value of the last **n** samples. **n**  is approximately equal to 1/(1-β). If β=0.9, then the averaging will be over last 10 samples. For bigger value of β, the result will be a much smoother curve as it 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 .

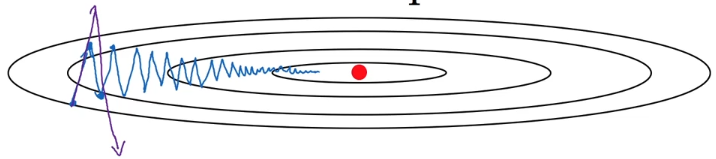
* + - **Bias correction in EWMA**



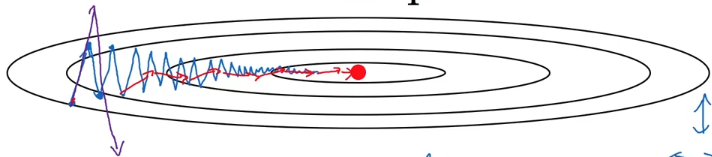
Purple curve is the EWMA and green curve is the EWMA with bias correction. The purple curves results due to the fact that the previous EWA term is assumed 0 for the first time. The bias correction tells us to divide the average term by , 

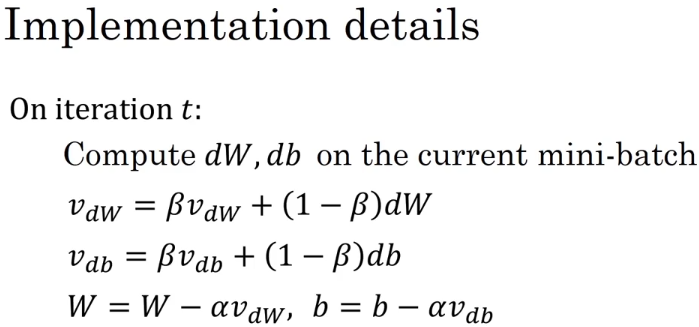
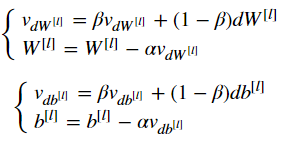
In this algorithm, the EWMA of the gradients is calculated and this averaged gradient is used to update the weights and bias. This algorithm almost always works faster when compared to the standard gradient descent algorithm.

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:

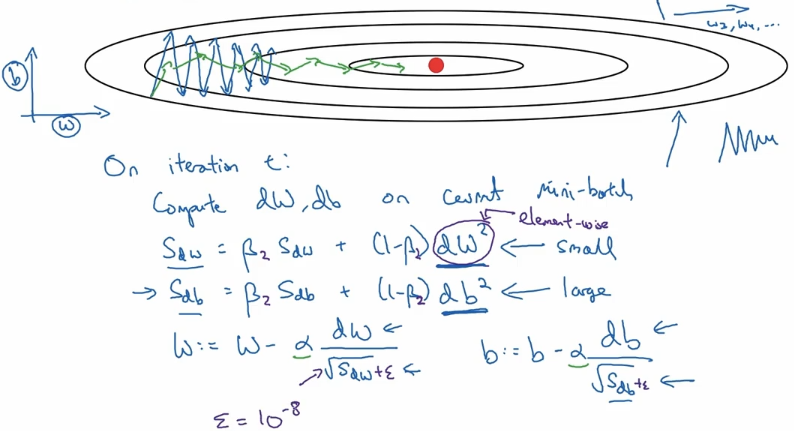
 The algorithm can reach the minimum much faster as there are less oscillation and also increases the chance of using higher learning rates.

or

β is the momentum and α is the learning rate.

* + **RMSprop**

RMSprop is similar to gradient descent with momentum with the difference being in the usage of square of the gradient instead of gradient.

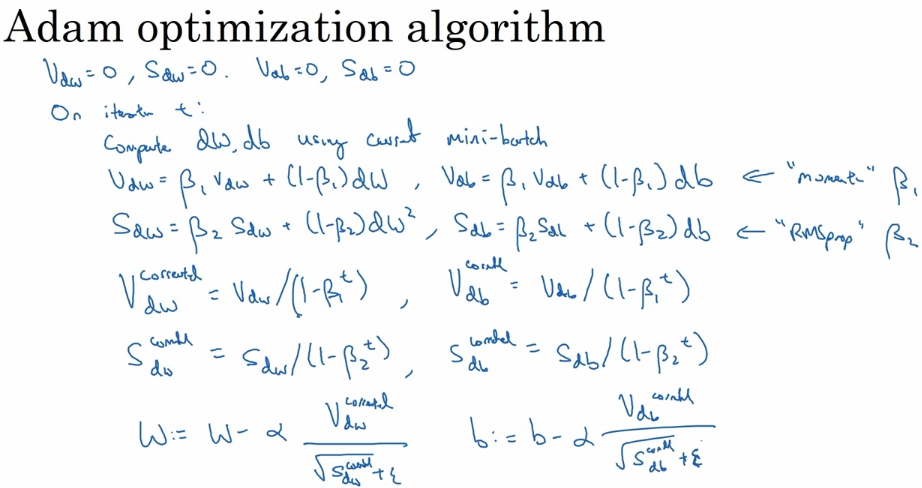


The EWMA is calculated on the square of the gradient. Then in the parameter update step, the square root of the EWMA of square of gradient is divided from the gradient. So if the slope(gradient) in one dimension is large, then its square is also going to be large. When the square root of this large value is divided, the update term becomes very small and the update in that dimension becomes slow. Similarly if the slope(gradient) in another dimension is small, then its square is also small. When the square root of this small value is divided, the update term becomes very large and the update in that dimension becomes faster. This way the oscillations are reduced and the minimum is reached faster. In other words, it  decreases the step for large gradient to avoid exploding, and increases the step for small gradient to avoid vanishing.

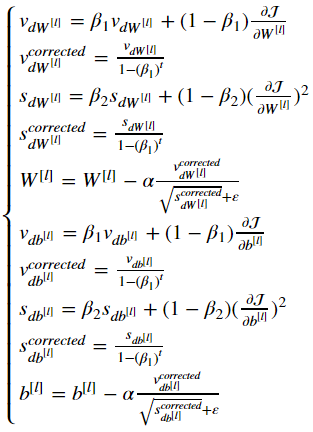
The axis labelled as ‘b’ and ‘w’, in the image, is only for representation purpose but it could be any parameter. So in the image, the ‘b’ has large gradient and ‘w’ has small gradient. To avoid divide-by-zero error, a very small number, ε, is added to denominator.

* + **Adam Optimizer**

This algorithm combines momentum with RMSprop. Typical Adam optimizer also includes bias correction.



or



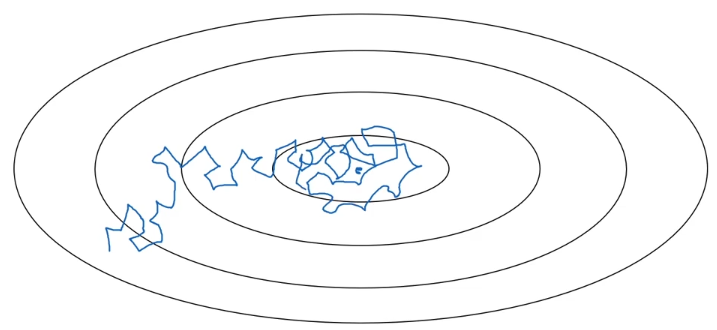
* t counts the number of steps take of Adam
* L is number of layers
* β1 and β2 are hyper-parameters that control the two exponentially weighted averages
* α is the learning rate
* ε is a very small number to avoid dividing by zero

First step inside the iteration loop is ‘momentum’ part and second step is ‘RMSprop’ part of the algorithm. Third and fourth step are bias correction on ‘momentum’ and ‘RMSprop’. Fifth step is weight updates using both ‘momentum’ and ‘RMSprop’.

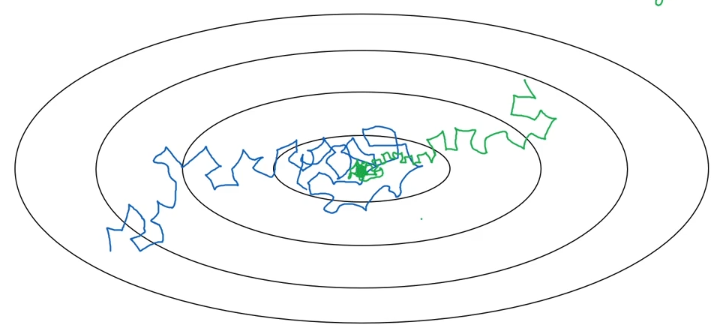
Adam is **Adaptive moment estimation**. β1 is used to compute the mean of the gradient which is called first moment and β2 is used to compute the mean of the square of the gradient and is called the second moment thus Adaptive moment estimation.

* + **Learning rate decay**

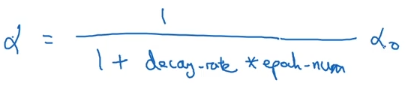
If we use a constant learning rate for all epochs and if the learning rate is huge(with the intension of decreasing the processing time of the training), then we will take big steps for updating the parameters for every iteration which might lead later to a lot of oscillations around the minimum but not actually reaching the minimum.

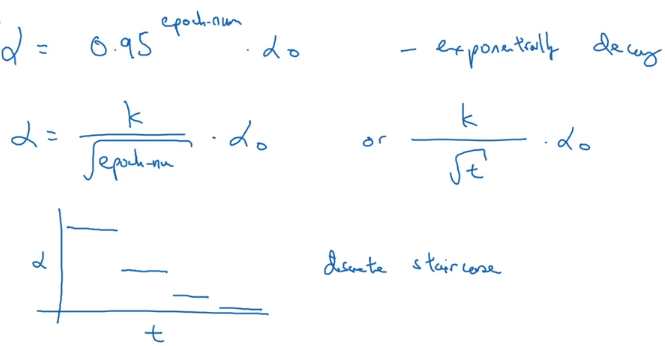


By using learning rate decay, we can still start with higher learning rate but keep reducing the learning rate every iteration so that we reach the minimum.



Below are some decay algorithms:

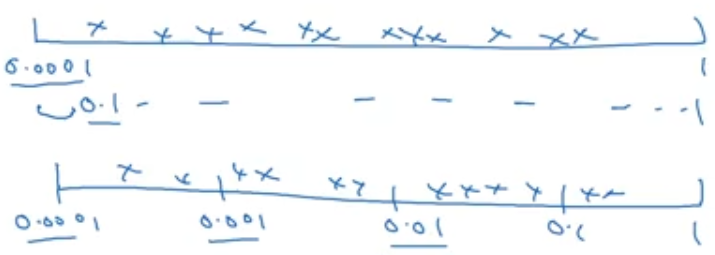




α0 is the initial learning rate. k is another constant. t is mini-batch number. The last method, discrete staircase, is to use discrete decreasing learning rate for a range of steps.

* **Hyper-parameter tuning**

A hyper-parameter value must be chosen uniformly from a range of values but care should be taken while considering the scale of the range.

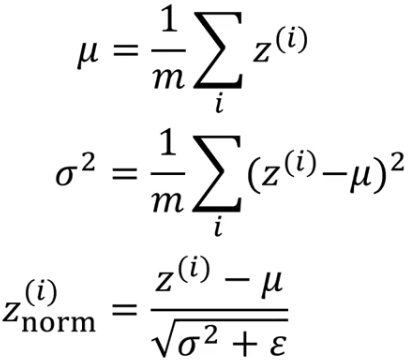


The first scale is linear and second scale is log.

If we want to choose the learning rate value from range 0.0001 to 1, using the normal linear scale might result in more number of values from 0.1 to 1 than 0.0001 to 0.1. So it is better to use log scale where the probability of getting values from 0.0001 to 0.001, 0.001 to 0.01, 0.01 to 0.1 and 0.1 to 1 is similar. Same holds true if we want to randomly sample value from 0.9 to 0.999 for β. For simplicity, 1-β is sampled from 0.0001 to 0.1.

* **Batch Normalization**

We saw how normalizing the input features, X, can help learning on a neural network and what batch norm does is that it applies that normalization process not just to the input layer but to the values even deep in some hidden layer. The normalization is done before applying the activation function.



The difference in normalizing the input features and hidden layers is that it is fine to normalize the input features to have zero mean and standard deviation 1 but the mean and standard deviation of the hidden layers depend on the activation function being used. If the activation function used is sigmoid and if the hidden layer has zero mean and standard deviation 1, then most of the values will fall in the linear portion of the sigmoid function and the neural network will not be able to make use of the non-linearity of the activation function.



To avoid the above situation, the normalized output is rescaled as follows:



γ and β are learnable parameters of the model. This way the model itself will learn to have the required mean and standard deviation.

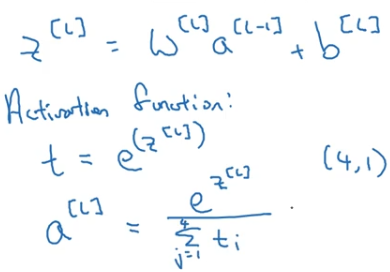
Batch normalization helps in dealing with the issue of “coherent shift”. When we use the batch gradient descent method, the mean and variance of each of the hidden layers will be similar every iteration as we are running through the same training dataset every iteration. But when we use mini-batch gradient descent, the mean and variance of output of each hidden layer will be different for different batches though they belong to the same dataset. Applying batch norm will make sure that the mean and variance of the output of each hidden layers is similar for different mini-batches.

Batch norm also has regularization effect. Each mini-batch is scaled by the mean and variance computed on just that mini-batch. This will be slightly different for every mini-batch. This adds some noise to the hidden layer output within that mini-batch. This is similar to dropout which can also be seen as adding noise to each hidden layer’s activation.

The μ and σ2 is calculated for every mini-batch during training time but during test time we may only have one test example and calculating μ and σ2 for that one test example doesn’t make sense. So an EWMA of μ and σ2 of all mini-batches during training time is calculated and that averaged μ and σ2 is used for normalizing the hidden layers during test time. Then the learned γ and β are used for rescaling the output.

* **Multi-class classification**

The generalization of logistic-regression is softmax regression that lets us recognize one of multiple classes. Softmax activation layer is applied to the final layer of a multi-class classifier model. It gives out the probability of each class as output.

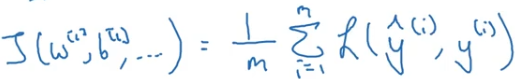


The term “softmax” comes from contrasting its method to “hardmax” method which would take the final layer output and map it to a vector of 1s and 0s  with the biggest element of the final layer mapped to 1 and to 0 for others. In contract, the “softamx” is a more gentle mapping from final layer to probabilities.

Below is the loss function for one training example with four classes:



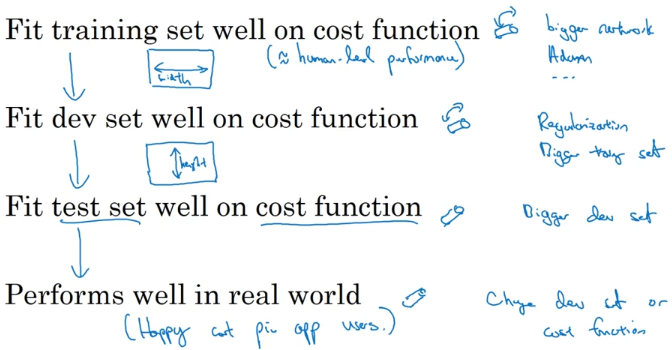
And the cost function for the entire(m) training example is:



**Structuring Machine Learning Projects**

* **Orthogonalization**

Orthogonalization in this context means being independent i.e. how to solve an issue without re-introducing the issue in an already solved issue.



Above is the chain of steps to follow if we decide to optimize our model.

First we have to make sure that the model is at least doing well on the training set. Then hope that it does well on dev set and later hope that it does well on test set too. Finally hope that the model does well on real world data.

If the model is not doing well on training set then we may think of using a **bigger network** or we may switch to a **better optimization** algorithm. If the model is doing well on training set and not fitting the dev set well then **regularization** can be used and using **bigger training set** also helps. If the model is doing well on both training and dev set but not on the test set then using **bigger dev set** will help. If the model is doing well on all the data(training, dev and test) sets but fails to deliver in the real world then change the dev set or the cost function.

* **Single number evaluation metric**

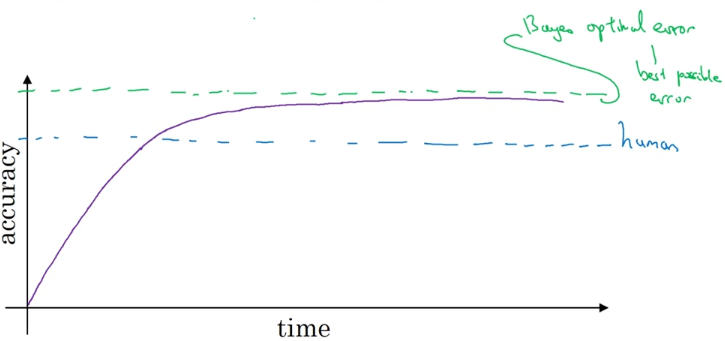
Precision and Recall does indicate how the model is performing. Precision is “off all the examples the model recognizes as positives what % are actually positives”. For e.g. if precision is 95% then it means that if the model says something is positive then there is 95% chance that it really is positive. Recall is “off all the examples which really are positives what % was correctly recognized by the model”. For e.g. if the recall is 90% then it means that off all the positives that are available the model was able to recognize 90% of them.

For easier and better understanding of the model’s performance the precision and recall are combined to give which is also the Harmonic Mean of precision and recall.

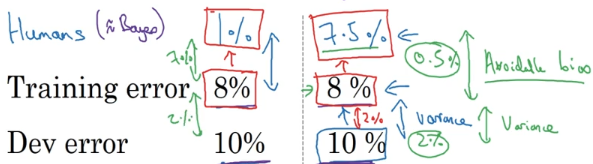
* **Size of train/dev/test set**

The old school machine learning thought of using 70/30% for train/test or 60/20/20% for train/dev/test is reasonable only if we have small size dataset like 100 or 1000 or 10000. If our dataset size is a million(1,000,000) or more, then using 98/1/1% is sufficient as 1% of a million is 10,000. The idea is to reserve more examples for training set.

* **Comparing to human level performance**



Bayes optimal error(or Bayes error in short) is the best possible error in the sense that there is no way any function can achieve an error lower than Bayes error. Baye’s error is, most of the time, not zero due the unavoidable noise in the data. We see above that the model’s(purple) accuracy increases considerably in the beginning and once it surpasses human accuracy there is not that much of an improvement. This is because human error is not that far from Baye’s error and in many cases we can assume human error ≈ Baye’s error. So we would want to keep improving the training error until we get down to human error but we don’t(can’t) actually want to do better than human error unless we are overfitting.



The difference between training and human error is called avoidable bias and difference between training and dev error is the variance. From the example on right, the avoidable error is 0.5% and variance is 2%, so it makes sense to focus on reducing the variance. From the example on the left, the avoidable bias is 7% and variance is 2% and in this case it makes more sense to focus on reducing the avoidable bias.

In summary, it is not always sensible to compare the training error to 0% because sometimes Baye’s error is non-zero and it is impossible for anything to reach Baye’s error. So we shouldn’t try to achieve 0% error on our model as we would be wasting time and resource on something that is impossible to achieve and instead we should focus on reducing the avoidable bias and variance.

* **Error Analysis**

Sometimes the dataset contains incorrectly labeled examples. If we consider training set, we don’t have to worry about these incorrectly labeled examples as deep learning algorithms are quite robust to **random errors**(random dogs are labelled as cats) in the training set. However, they are not robust to systematic errors(all white dogs are labelled as cats) as the DL algorithm will learn to classify white dogs as cats.

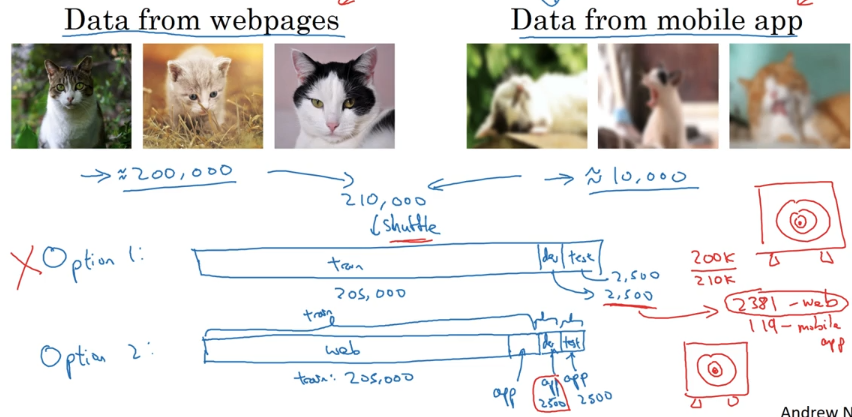
When we consider dev set, deciding on whether to invest time on correcting those errors depends on three numbers: Overall dev-set error(e.g. dev-set error is 10%), errors due to incorrect labels(e.g. 6% of dev-set errors are due to incorrect labels, then 6% of 10% is 0.6%) and errors due to other causes(the remainder, 10-0.6=9.4% of errors are due to other causes such as mis-recognizing dogs, great cats and blurry images). This shows that investing time in resolving mis-labeled images is not worth as the error contribution is only 0.6%. It makes more sense to focus on reducing the error due to other causes which is a significant number, 9.4%.

Now if we consider dev-set error to be 2% and errors due to incorrect labels is still 0.6%(which is 30% of 2%) then it is worthwhile to fix those mis-labelled examples.

The main purpose or goal of the dev-set is to help us to select between two classifiers A and B. Let’s say A has 2.1% and B has 1.9% dev error but we can’t decide which classifier is better as both have 0.6% error due to mis-labelled examples in dev-set. So this is a good reason to go ahead and fix those mis-labelled examples.

* **Training and testing on different distribution**

There might be cases where the dataset in hand has a lot of examples of a particular distribution and the remaining small set of examples have a different distribution which actually match the real world examples. For e.g. for a cat classification app for mobile, you have 200,000 images of cats from the web which are clean and of good size and resolution and 10,000 images of cats taken from mobile camera with low resolution.

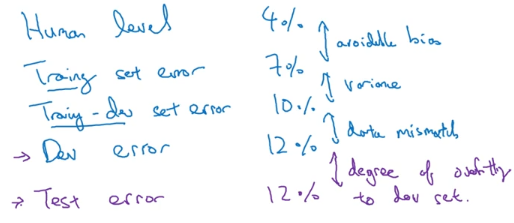


If we now simply follow the fact that the training and testing sets should be of same distribution then we would go about shuffling the data so that both training and testing sets have data from same distribution. If we assign 2,500 examples each for dev and test set then the training set would have 205,000 examples. This kind of splitting has a huge disadvantage due to the proportion of the images taken from web and mobile. Out of the total images, the mobile images make only 4.76%. This would mean, out of the 2500 examples in the dev set only 119 are from mobile images which implies that we are giving the model a target for optimization which is different from the real world target. The real world target is to classify cats in images taken from mobile while we are providing a lot of web images of the cat in the dev set.

The other option is to have all 200,000 web images + 5,000 mobile images in the training set and the dev and test set to have all mobile images with each 2500 images. The advantage of this split is that we are providing a target to the model for optimization which is same as the real world target. The disadvantage is that the training distribution is different than the dev/test distribution but this split will get better performance in the long term and there are specific techniques to deal with training set coming from different distribution than dev/test distribution.

* **Bias and Variance analysis with mismatched data distribution**

With different distribution of data in training and dev set, it becomes difficult to conclude whether the difference in training error and dev error is either because of variance or simply due to data mis-match. So to better differentiate these two effects, a new dataset is introduced which is called training-dev set. This dataset will have same distribution as training set but is not used for training(i.e. not used in the backpropagation algorithm).



* **Transfer Learning**

Taking the knowledge that the neural network has learned from one task and applying that knowledge to a separate task. For e.g. If we have a neural network that can recognize cat images, then that knowledge or part of the knowledge can be used to diagnose X-ray images. The way to implement this is by deleting the last layer or last few layers along with its weights and adding new layers with randomly initialized weights and retraining the new layers with X-ray images. If there are less X-ray images then we may want to retrain only the newly added layers but if we have a enough X-ray images then we can retrain the entire network. In transfer learning terms, the training on cat images will be called pre-training and later training on x-ray images will be called fine-tuning.

Transfer learning makes sense when we have lot of data for the problem we are transferring from and usually relatively less data for the problem we are transferring to. So for e.g., let's say we have a million examples for image recognition task which is a lot of data to learn a lot of low level features or to learn a lot of useful features in the earlier layers in neural network. But for the radiology task, maybe we have only a hundred examples. So a lot of knowledge we learn from image recognition can be transferred and can really help us to get going with radiology recognition even if you don't have all the data for radiology.

In summary, transfer learning(from task A to task B) makes sense when:

* Task A and B have the same type of input. E.g. transfer learning from cat image recognition to X-ray image diagnosis. General speech recognition to wake words(‘OK Google’, ‘Hey Siri’, ‘Alexa’) recognition
* There is lot more data for task A than task B
* Low level features from A could be helpful for learning B
* **Multi-task learning**