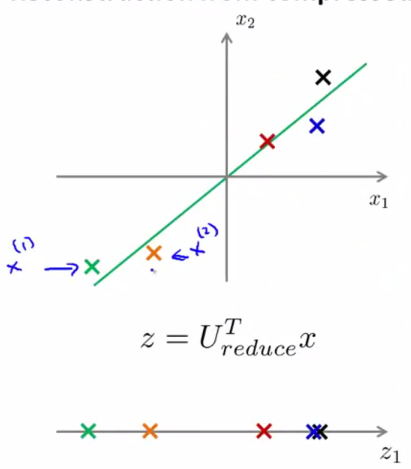
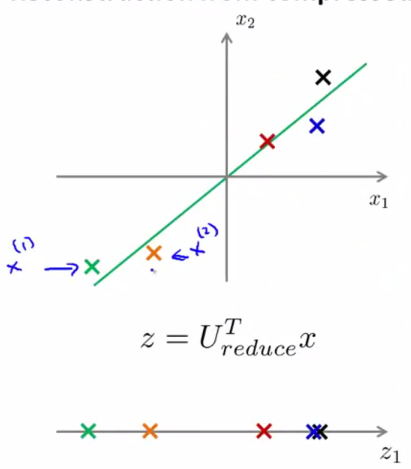
***Dimensionality Reduction***

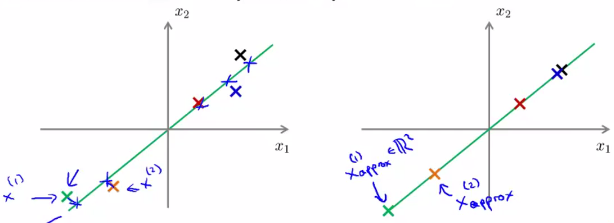
***Applying PCA***

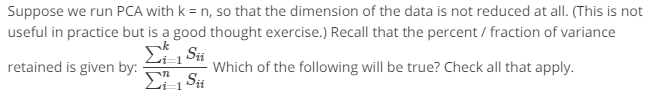
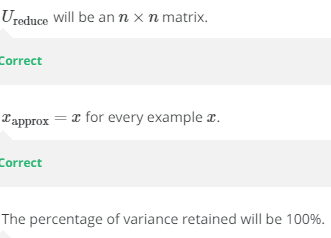
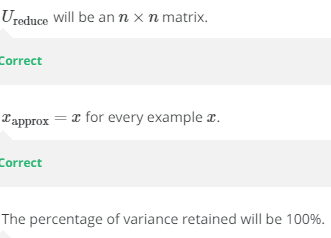
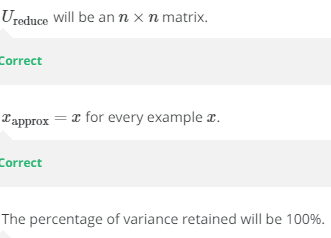
**I. RECONSTRUCTION FROM COMPRESSED REPRESENTATION**

* Since PCA is a compression algorithm, there should be a way to go back from the compressed representation back to an approximation of the original high-dimensional data.
* So given a 100-dimensional z(i), how do you go back to your 1000-dimensional original representation, x(i)
* In the PCA algorithm for 2D data, we have examples + project them onto a 1D surface + then now use a real number, say z1, to specify the location of these example points after they've been projected onto this surface.



* So, given the point z1, which is in Ɍ, how can we go back to this original 2D space/map this z1 back to some approximate representation x in Ɍ2 of whatever the original value of the data was?
* Whereas z = U(reduced)(t)\*x, if you want to go in the *opposite* direction, we write:
* **x(approx) = U(reduced)\*z**
* Remember U(reduced) is an n\*k-dimensional vector + z is a k\*1-dimensional vector, so when you multiply these, x(approx) is going to be an n\*1 or n-dimensional vector.
* The intent of PCA is if the squared projection error is not too big, this x(approx) will be close to the original value of x used to derive z in the first place.



* This is a pretty decent approximation of the original data
* This process = **reconstruction** of the original data (think of trying to reconstruct the original value of x from the compressed representation)
* 
* 

**II. CHOOSING THE NUMBER OF PRINCIPAL COMPONENTS**

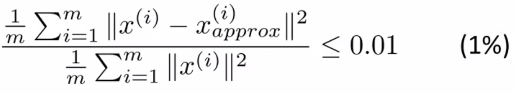
* So, in the PCA algorithm we take n dimensional features + reduce them to k dimensional feature representation, + this number k is a parameter of the PCA algorithm.
* k = the **number of principle components** **retained**
* We know PCA tries to minimize the average squared projection error (difference between original data X + projected version, X(approx)(i)) on the lower-dimensional surface.



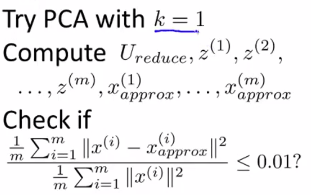
* 1:01
* **Total variation** in the data is the squared average length of examples X(i)



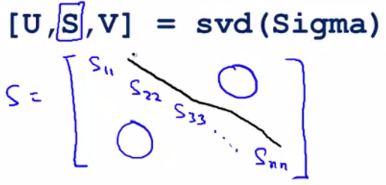
* "On average, how far are my training examples from the origin/the vector of all 0’s"
* A pretty common rule of thumb for choosing k is to choose the smallest value such that the ratio between average squared projection error and total variation is less than 0.01.



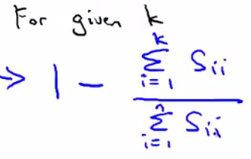
* The way most people think about choosing k is, rather than choosing k directly, being concerned with that this ratio is
* If it is 0.01, in the language of PCA we say “99% of the variance is retained“
* If using PCA + want to tell someone how many principle components you've retained, it’s more common to say “I chose k so that 99% of the variance was retained”
* "I had to 100 principle components" or "k was equal to 100 in a 1000-dimensional data" is harder to interpret
* Other common values is 0.05, or 5%, + you’d say “95% of the variance is retained”
* As low as 85% of variance contained would be a fairly typical range in values, while 95-99 is the most common range of values people use.
* For many data sets, you can often reduce the dimensions of the data significantly + still retain most/99% of the variance.
* B/c for most real-life data sets, many features are highly correlated, + it turns out to be possible to compress the data a lot + still retain 99%/95% of the variance.
* So how do you implement this? Here's 1 algorithm you might use



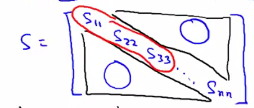
* Start off w/ k = 1, run through PCA (compute U(reduced), z1, z2, up to z(m), x(approx)1 up to x(approx)(m), then check if 99% of the variance is retained.
* If it isn't, try k = 2 and repeat
* This procedure seems horribly inefficient
* Fortunately when you implement PCA it actually, the inner computation step actually gives us a quantity that makes it much easier to compute the average squared projection error + total variance
* When you're calling **SVD** on the covariance matrix δ to get the matrices U, S, V, it also gives back a matrix S, an diagonal n\*n square matrix



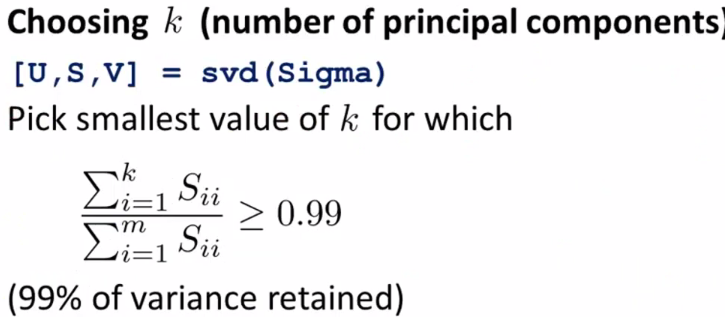
* S(1,1) to S(n,n) are the only non-zero elements of this matrix
* It turns out that for a given value of k, our PCA ratio can be computed much more simply as:

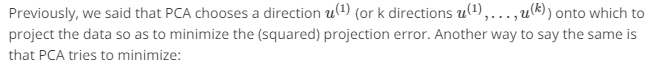


* If k = 3, we compute the sum of S(i,i) for i = 1 through i = 3



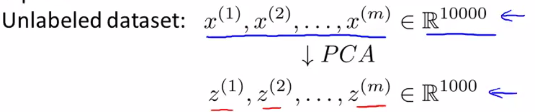
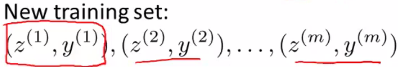
* For the denominator, we sum all the diagonal entries.
* Then we test if 1 – this ratio is less than or equal to 0.01 to be sure 99% of the variance is retained.
* Can then just slowly increase k + test this quantity to find the smallest value of k that ensures 99% of the variance is retained.
* If you do this, you only need to call SVD()once b/c it gives the S matrix
* Once you have the S matrix, you can then just keep doing this calculation by increasing k
* This procedure is much more efficient + allows you to select k w/out needing to run PCA from scratch over + over.
* Run SVD once to get all those diagonal numbers + vary k in the ratio to find the smallest value of k so that 99% of the variance is retained.



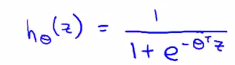
* Even if you picked k manually, if you want to explain to others what you did, a good way to explain the performance of your implementation of PCA is to compute this quantity to find the percentage of variance retained.
* If you report that number, people familiar w/ PCA get a good understanding of how well your k-dimensional representation is approximating the original data set
* 99% of variance retained = a measure of **squared reconstruction error**
* A ratio <= 0.01 gives people a good intuitive sense of whether your implementation of PCA is finding a good approximation of the original data set.
* If you apply PCA to very high-dimensional data sets, very often, just b/c data sets tend to have highly correlated features, PCA will be able to retain 99% of the variance, even while compressing the data by a very large factor.
* 
* 

**III. ADVICE FOR APPLYING PCA**

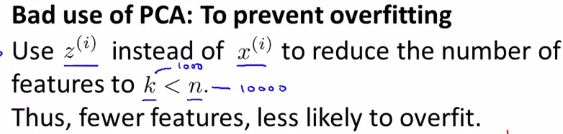
* We know PCA can sometimes be used to speed up the running time of a learning algorithm.
* Say you have a supervised learning problem w/ inputs X and labels Y + our examples X(i) are very high-dimensional feature vectors (say 10k, like in CPU vision problems w/ 100x100 images, which means 10,000 pixels = X(i) = feature vectors containing 10,000 pixel intensity values)
* W/ very high-dimensional feature vectors, running a learning algorithm can be slow
* If you feed 10,000 dimensional feature vectors into logistic regression, NN, or SVM, it’s lot of data + can make your learning algorithm run more slowly.
* Fortunately, w/ PCA, we can reduce the dimension of data to make algorithms run more efficiently.
* 1st check the labeled training set + extract *just the inputs* X, which give us an unlabeled training set X1-X(m) in Ɍ10,000
* Then apply PCA to get a *reduced* dimensional representation of the data (maybe 1k-dimensional feature vectors, a 10x savings)
* This gives us a *new* training set w/ Z’s paired w/ Y’s instead of X

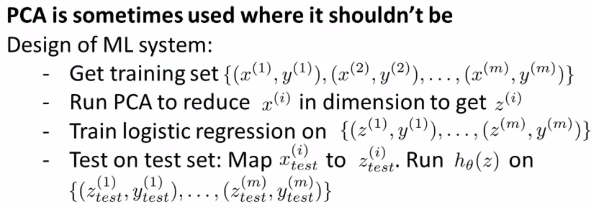
* Can take this reduced-dimension training set + feed it to a learning algorithm to learn the hypothesis **hϴ(z)** to make predictions.
* Ex: Logistic Regression:



* If we have a new test example X, take it + map it through the same mapping that was found by PCA to get the corresponding Z to feed to the hypothesis which makes a prediction on test input X.
* What PCA does is define a mapping from X to Z which should be defined by *running PCA only on the TRAINING set*
* This mapping that PCA is learning will compute the set of parameters (w/ feature scaling + mean normalization) + also computes matrix U(reduced).
* These are all parameters learned by PCA + we should be fitting parameters *only to training sets* + NOT to CV or test sets
* U(reduced) + the like should be obtained by running PCA *only on your training set*
* Having found the parameters on the training set, you can then apply this mapping to other examples in your CV or test sets
* For many problems, we can reduce dimensional data by 5x-10x + still retain most of the variance while barely effecting performance in terms of classification accuracy of a learning algorithm
* By working w/ lower-dimensional data, our learning algorithm can often run much faster.
* To summarize applications of PCA:
* Compression
* Reduce memory/disk space needed to store data or to speed up a learning algorithm.
* In these applications, in order to choose k (dimension to reduce to), we figure out the **% of variance retained** + hope to retain 99% of the variance
* Visualization
* Since usually we only know how to plot 2D or 3D data, for visualization applications, we'll usually choose k = 2 or k = 3, b/c we can plot only 2D and 3D data sets.
* There is often 1 frequent misuse of PCA you sometimes hear about: *to try to use it to prevent over-fitting.*
* This is not a great way to use PCA
* If we have Xi + n features, if we compress the data + use Zi w/ reduced features k, then we have a much smaller number of features + maybe we're less likely to over-fit
* It's not that PCA works badly in this application



* If you want to use PCA to reduce the dimensional data to try to prevent over-fitting, it might actually work OK, but this just is not a good way to address over-fitting
* Instead, if worried about over-fitting, use **regularization** instead
* If you think about how PCA works, it does not use the labels Y 🡪 only looking at inputs Xi + using that to find a lower-dimensional approximation to data
* It throws away some info/reduces the dimension of data w/out knowing what the values of Y is,
* This is probably okay if, say, 99% of the variance is retained while throwing away some valuable info.
* If retaining 95%-99% of the variance, just using regularization will often give you at least as good a method for preventing over-fitting + will often just work better
* When applying linear or logistic regression, or some other method W/ regularization, THE minimization problem actually *knows* what values of Y are + therefore so is *less likely to throw away valuable info*
* PCA is good if your main motivation is to speed up a learning algorithm, but not to prevent over-fitting
* Using regularization instead is really what many people would recommend doing instead.
* PCA is a very useful algorithm, but there’s 1 last misuse where it shouldn't be used



* Before writing down a project plan like this, which incorporates PCA, 1 very good question to ask is, “What if we just do the whole without using PCA?”
* Often people do not consider this before coming up w/ a complicated project plan + implementing PCA
* Before implementing PCA, first, look at whatever you want to do + consider doing it w/ original raw data Xi
* Only if that *doesn't do what you want*, *THEN* implement PCA + use Zi.
* Before using PCA + reducing the dimension of the data, just train a learning algorithm on original, raw data inputs Xi
* Only if you have a strong reason to believe/strong evidence *that* doesn't work (learning algorithm ends up running too slowly or the memory/disk space requirement is too large + you want to compress your representation) should you implement PCA + use a compressed representation.
* Sometimes people start off w/ a project plan that incorporates PCA inside, but whatever they're working on will work just fine even w/out using PCA
* Just consider that as an alternative before you go to spend a lot of time to implement PCA in, figure out what k I, etc.
* 
* Despite these last sets of comments, PCA is an incredibly useful algorithm when you use it for the appropriate applications:
* Speed up running time of learning algorithms
* Compress data
* Reduce the memory/disk space requirements
* Visualize data.
* PCA is 1 of the most commonly used + 1 of the most powerful unsupervised learning algorithms.