DIMENSIONALITY REDUCTION : AUTOENCODER VS PCA

May 6, 2020

Rohit Gangurde
Boise State University

Contents

0.1 Introduction

In this day and age, global data has increased ten folds. This rapid generation of data brings with it the problem of *dimensionality*. The human brain is capable of visualizing data which is 1, 2 or 3 dimensional, anything above that and we hit a roadblock. Data today can have more than 100 dimensions. For one of my projects, the data that I was using had **276** *dimensions* ie. *features*. Commonly known as **Big Data**, these dimensions makes visualizations, computations really difficult. This problem of *dimensions* is officially called **the curse of dimensionality**. An article on DeepAI[1] states that *the curse of dimensionality* refers to a phenomena that occurs when **classifying, organizing and analyzing high dimensional data** that does not occur in low dimensional space, specifically **the issue of data sparsity and closeness of data**.

To overcome *Big Data problems* and *the curse of dimensionality*, we have several **dimensionality reduction** techniques. (Meyer-Baese & Schmid, 2014) [2] defines *dimensionality reduction* as **the process of reducing the number of random variables or attributes under consideration**. The *random variables or attributes* here refers to **the features** in our dataset, also known as **the columns** of our dataset. These techniques reduces the number of features in our data. This reduction in dimensions help us visualize our data and makes it easier to identify trends and patterns. How do they achieve this? **Linear Algebra**. In how many ways can we achieve this? There are **12** common dimensionality reduction techniques. We won't be looking at all of them in this paper. For this paper, we are focus on two distinct techniques, **Principal Component Analysis (PCA) and Autoencoders**.

Principal Component Analysis

PCA was originally invented in 1901 by Karl Pearson in the field of mechanics. It was then independently developed by Hotelling (1933)[3] in 1933. *PCA* has been employed in different fields and it is also named differently. Principial Component Analysis that we know of today maps the data from original space to a new space while retaining most of the information.

Autoencoders

Autoencoders were first introduced by (Rumelhart, Hinton, & Williams, 1986)[4]. Originally *autoencoders* were proposed as a method for unsupervised pre-training. But as the size of the data has grew over the years, *Autoencoders* have found new applications in dimensionality reduction. It has grown popular over the years due to its ability to model *non-linear functions*.

Both *PCA* and *Autoencoders* serve the purpose of **dimensionality reduction**. But the way they achieve that is totally different. The transformations that they are capable of performing are different. This paper will be investigating the *dimensionality reduction* mechanism and capabilities of **Principal Component Analysis and Autoencoders**.

0.2 Formulation

Let's take a look under the curtain and explore the *Linear Algebra* behind these two techniques and understand how different they are in their approach to dimensionality reduction.

0.2.1 Principal Component Analysis

Strang[5] mentions in his book *Linear Algebra and Learning From Data* that the principal components of matrix $A \epsilon \mathbb{R}^{mxn}$ are its **singular vectors**. **PCA** makes use of the *largest singular values* σ*'s* connected to the first columns u*'s* and v*'s* of the orthogonal matrices U and V. *Singular Value Decomposition* allows us to breakdown A into,

$$A_k = \sigma_1 u_1 v_1^T + ... + \sigma_k u_k v_k^T \qquad \textbf{rank}(A_k) = \textbf{k}$$

This gives us the closest approximation we can get to A that is A_k. This special **rank k matrix** as *the best choice* for representing A was proved by **Eckart-Young** in 1936 using *the Forbenius Norm* which we use later on. Strang states that the key point of *PCA* is to find a line such that it passes through all the data points with mean zero and the line should be close to $(0,0)$. This line is usually in the *direction* of **the first singular vector** u_1 of A.

One of the first things to do for *PCA* is to *center the data*, that is all the rows of A should add to **zero**, which helps us in obtaining the *covariance matrix* of A which is yet another crucial part of *PCA*. The sample covariance matrix of A is given by

$$S = \frac{AA^T}{n-1}$$

The **covariance matrix** is used in calculating the *total variance* which is how we know how much fraction of the information the principal components account for. The total variance is computed using the Forbenius norm.

$$\textbf{Total Variance } T = \frac{||A||_F^2}{n-1} = \frac{||a_1||^2 + ... + ||a_n||^2}{n-1}$$

This equation represents the trace of the covariance matrix S. The **trace of S** is the sum of the diagonal. The diagonal of the covariance matrix contain *eigenvalues*. So, the trace can be written as,

$$\textbf{Total Variance } T = \frac{\sigma_1^2 + \sigma_2^2 + \ \ ... \ \ + \sigma_r^2}{n-1}$$

The *core* connection of *Principal Component Analysis* to *Linear Algebra* are the **singular values** and the **eigenvalues**. Coming back to the information representation, the first principal component u_1 explains $\frac{\sigma_1^2}{T}$ of the total variance. Each component tries to explain as much information as it can.

So, to calculate **the principal components**, we need to :

1. Compute the covariance matrix of the data
2. Compute the Eigenvectors and the Eigenvalues using the covariance matrix
3. Get the k Eigenvectors with the largest eigenvalues
4. Form a matrix T of d x k dimensions using the Eigenvectors of k largest Eigenvalues
5. Transform the dataset by applying the matrix T to it

0.2.2 Autoencoders

Autoencoders are **neural networks** which learn the *mapping* of the **input** to the **input**. It can be a simple feed-forward neural network or can be a complex neural net with a deep architecture. The *architecture* of the **autoencoders** is specific to the *data* it's trying to model.

An autoencoder can be easily split between two parts, **the encoder** and **the decoder**. We can *encode* the data by *reducing* the number of units in each subsequent layer of the neural net until we get to the desired **k** units. This will be the **encoder layer** with k units. After this we can build back up by *adding* more units in each subsequent layer until we reach to the original input dimensions, this is our **decoder layer**.

The key element of an autoencoder architecture is its **activation function**. The choice of activation function depends on the *data* and the *purpose* of the autoencoder. Each layer can have a different activation function, choice of which depends on the purpose of the layer. The activation functions influence the encoded representation of the data as the neural net decides how the data is combined. If a *linear* activation function is used for each layer, then the encoder layer of the autoencoder will *directly* correspond to *principal components* obtained from **Principal Component Analysis**.

The paper (Baldi & Lu, 2011)[6] does a great job of explaining autoencoders and their inner workings. To simplify, the *mechanism* for the autoencoders is made up of **two parts**, *encoder* and *decoder*.

$$\phi : \chi \longrightarrow F$$

$$\psi : F \longrightarrow \chi$$

The encoder part uses the *encoder function* ϕ to **map** the original data space χ to the *encoded* space F. The decoder part uses the *decoder function* ψ to **map** the data from the *encoded* space F to the *original* data space χ.

Given an autoencoder with **one** hidden layer, the *encoder function* can be written as $h = \sigma(Wx+b)$, where h is the *encoded representation* of the data, σ is the *activation function* such as **sigmoid** or **rectified linear unit**, W is the *weight matrix*, x is the *original* data and b is the *bias vector*. The weight matrix and the bias vector are updated for each feature *iteratively* using **partial derivatives**

and **backpropogation**. The *decoder function* can be written as $x' = \sigma'(W'h + b')$, where x' is the *decoded/reconstructed* representation of x and W', σ' and b' may not correspond to the previous W, σ and b. As is evident from the equations, the activation function σ plays important role in *mapping* the data from its *original* space to an *encoded* subspace and vice versa. Autoencoders try to *minimize* the **loss function** $L = f(x, x')$ such that it *penalizes* x' for being **dissimilar** to x.

Strang[5] states on page 397 that Deep Learning is fundamentally a giant problem in optimization of the learning function F. We also have to *minimize* the loss function L for which we keep *updating* weights (matrix W) and the bias (vector b). To *minimize* the loss function, we *compute* the partial derivatives of L with respect to the weights, the result of which should be *zero*. **Gradient descent** tries to *compute* the derivatives of the **learning function** F, which leads to $\partial L/\partial x$. This helps in *updating* **weights** and **bias** terms each iteration and this is where *Backpropogation* comes in. **Backpropogation** is a method of quickly computing derivatives using chain rule which we will not discuss for the purposes of this paper.

0.2.3 Comparison

The *biggest* difference between autoencoder and pca is the **transformation** they are capable of performing. PCA can only learn *linear* transformations that project data into a subspace, where the *vectors* are defined by *variance* in the data. Autoencoders on the other hand are capable of transforming the data **linearly** or **non-linearly** with the use of *activation functions*. Because autoencoders can reconstruct data by mapping to subspace, it encodes the important features into the latent subspace.

The *principal components* obtained from PCA have **no** *correlation* to each other, which helps when using some machine learning algorithms. The encoded features obtained from autoencoders might have **some** *correlations* between them.

Principal Component Analysis aims to find a compact representation of matrix A such that k **principal components** explain *almost all* of the information of A. Whereas, autoencoders aim towards *accurate reconstruction* of the original data.

0.3 Results

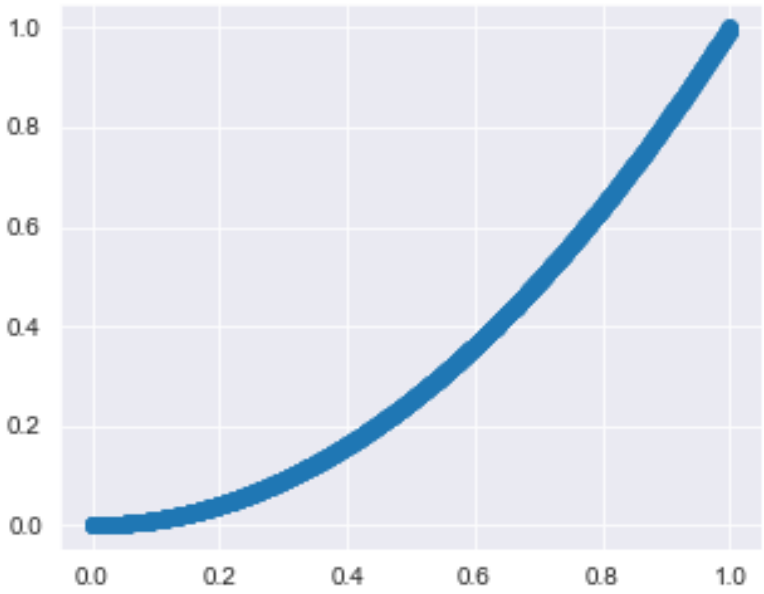
For comparing autoencoders and pca, I decided to work with two datasets. The datasets are :

1. Data generated from the equation $y = mx^2 + c$
2. House Prices : Advanced Regression Techniques from Kaggle

The first dataset will be used to demonstrate the **non-linear modelling** capabilities of the two methods. The second one is used for demonstrating **dimensionality reduction** capabilities.

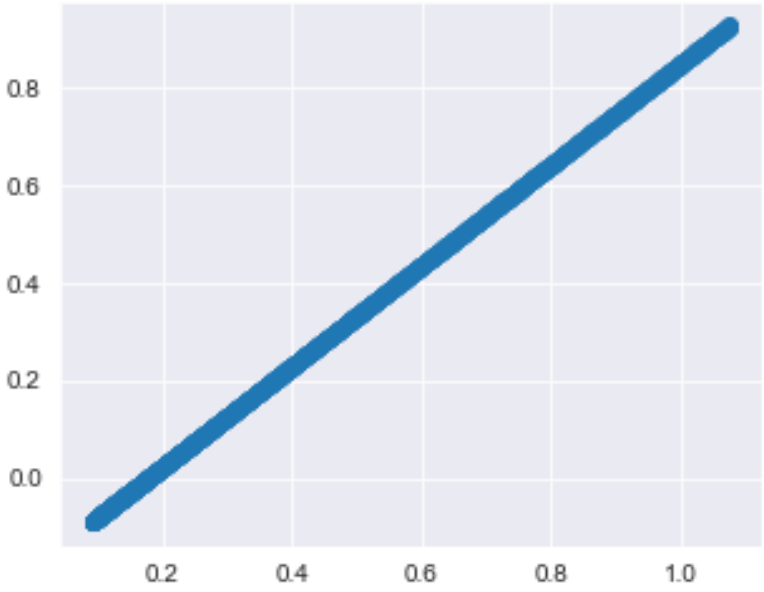
0.3.1 Non-Linear Reconstruction

One of the most classic equations of line is $y = mx + c$, but we are going one step ahead and analyse the performance of pca and autoencoders on the equation $y = m^2x + c$. The plot below shows the datapoints generated from that equation,

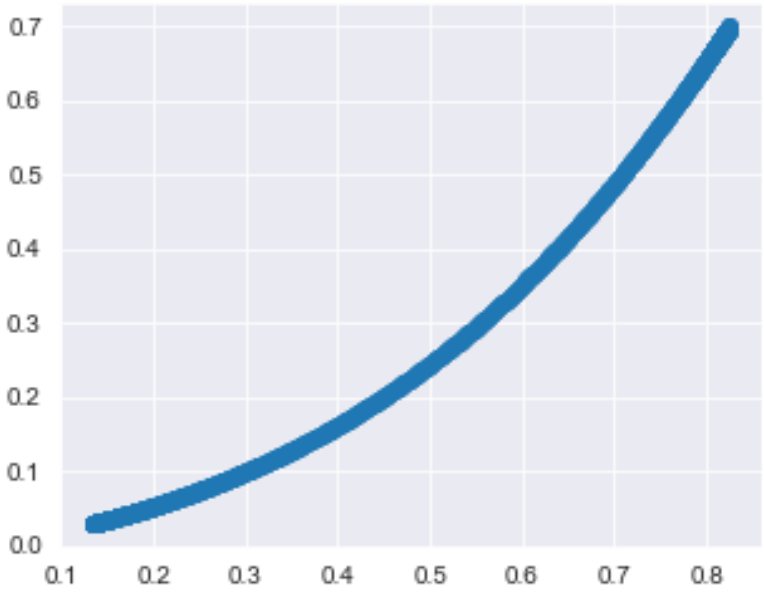


We can see that the relationship between x and y is non-linear which is evident by the curve. I had mentioned that one of the shortcomings of PCA is that

they don't model well to non-linear transformations, which is potrayed by this plot,



I used the *scikit-learn* library to perform pca on the data and supplied 1 to the parameter **n_componets**. The reconstructed data has completely **lost** the *non-linear* relationship between the data points. When I used autoencoder on the same data, the results were different,



Autoencoders do a fine job of modelling non-linear transformations. Though this is not the most accurate reconstruction of the data, but that is because of the model architecture. I am using 7 layers : 1 input, 1 output, 4 hidden, 1 encoder layer. The *activation function* used on the hidden layers is **linear**. The encoder layer and the output (decoder) layer uses **sigmoid** *activation functions*. The model is compiled with **adam** *optimizer* and uses **mean squared error** as the *loss function*. This architecture isn't the best of course, but does good enough job in demonstrating the fact that autoencoders do a better job at modelling *non-linear* transformations than pca.

Source code can be found in the appendix

0.3.2 Dimensionality Reduction

The goal of the last section was to display the inability of pca to model non-linear transformations. This section focuses on the *dimensionality reduction* part. The dataset used for this section can be downlaoded from **Kaggle**. The dataset contains 80 columns/features. Once all the *preprocessing* is done and the *categorical* features are converted to *numerical*, the number of features

quickly go up to 276, that's because *one hot encoding* was used to convert the categorical features. The data is also *standardized* before transformation (Data already processed). It is difficult to visualize 276**dimensions**. It is difficult to project data in such a high dimensional space, therefore we use **dimensionality reduction**. I reduce the dataset to 2 dimensions. This is a huge reduction resulting in *loss* of information. The autoencoder architecture used for this data is different than the last one. Because autoencoders are *trained* on data, it is difficult to have a generalized model for different types of data. I used *scikit learn's* PCA module to fit and transform this data. The resulting plot from PCA encoding is this,

There are a few details to point out here. The **red** data points are points corresponding to *Sale Price* feature being **less** than its **mean** value and the **green** data points are points corresponding to *Sale Price* feature being **more** than its *mean* value. We can see that there isn't quite good separation between the two sets of points. But value 0 on the x-axis can be used as a separator between the **two** "clusters" as 0 is the *mean* of the dataset due to *standardization*. Now let's look at the autoencoder transformation.

So we can see that the two plots are **distinct** from each other. Again, the *red* and *green* data points are the same as last time. We can see that the *green* points are in **two** clusters. There are *two* **arcs** of the *green* points. The *red* data points are spread across the axes. Again, the *seperation*of the data points is not quite distinct. The *scale* of the encoded data points is **not** quite *normal*. That's because the task of the autoencoder is to *reconstruct* the input, so anything in middle might **not** be the most *accurate representation* of the data. The autoencoder model used for this data is deeper than the one used for previous data. This autoencoder has 13 layers : 1 input, 1 output, 10 hidden, 1 encoder layer. Same as last time, *encoder* and the *output* (decoder) layer uses **sigmoid** *activation function* and the *hidden* layers use **linear** *activation function*. The model is compiled with **rmsprop** *optimizer* and with **mean squared error** as the *loss function*.

Source code can be found in the appendix

0.4 Conclusion

Going through the results, it is *evident* that there are advantages and disadvantages in using both the methods. The principal component analysis is a more *theoretical* linear algebraic approach which does a good job in *retaining* information of the original data. The **eigenvalues** and **eigenvectors** play a big role in *retaining* the information and *mapping* it to a *smaller latent* data space. The autoencoders on the other hand are *neural networks* which *learn* the *mapping* from the *input* to the *input*. So they are **not** focusing on *retaining* information from the original data, they are just looking for *patterns* in the data to *map* it from original data space to lower dimensional space and back to the original data space again. So both the methods differ in its *core* ideology.

Autoenecoders are computationally *expensive* than principal component analysis and are also difficult to create as there are many configurations one can try. But with proper architecture and optimizers, autoencoders can most of the time perform *better* than principal component analysis. Also, principal component analysis is a **method** of performing dimensionality reduction whereas autoencoders are **a family of methods** so it is more than capable of outperforming principal component analysis.

Principal Component Analysis is a good method to use when you want *quick* results and when you know that your *data* is **linear**. It is guaranteed to provide *components* which *retain* most of the **variance** of the data. Autoencoders are time consuming to develop and often require fine tuning the hyperparameters. Both of them have their perks and are useful in many scenarios.

0.5 References

[1] Curse of Dimensionality. url: *https://deepai.org/machine- learning-glossary-and-terms/curse-of-dimensionality.*

[2] Volker Schmid Anke Meyer-Baese. (2014). *Pattern Recognition and Signal Analysing Medical Imaging (Second Edition).* Academic Press. isbn: 978-0128101162.

[3] Harold Hotelling.(1933) Analysis of a Complex of Statistical Variables Into Principal Components. *Journal of Educational Psychology 24.*

[4] Rumelhart, D., Hinton, G. Williams, R. (1986). Learning representations by back-propagating errors. *Nature 323, 533–536.* https://doi.org/10.1038/323533a0

[5] Glibert Strang.(2019). *Linear Algebra and Learning Data.* Cambridge Press. isbn: 978-0629219638-0.

[6] Zhiqin Lu Pierre Baldi.(2011). Complex-Valued Autoencoders. *Neural Networks.*

0.6 Appendix

1. Clean Housing Data : View it on Github
2. Data Preprocessing : View it on Github
3. Non - Linear Transformation : View it on Github
4. Dimensionality Reduction : View it on Github