Elior Cohen Follow
Data scientist, Pythonista
Jul 2, 2017 · 13 min read

Reducing Dimensionality from Dimensionality Reduction Techniques

In this post I will do my best to demystify three dimensionality reduction techniques; PCA, t-SNE and Auto Encoders. My main motivation for doing so is that mostly these methods are treated as black boxes and therefore sometime are misused. Understanding them will give the reader the tools to decide which one to use, when and how.
I'll do so by going over the internals of each methods and code from scratch each method (excluding t-SNE) using TensorFlow. Why TensorFlow? Because it's mostly used for deep learning, lets give it some other challenges :)
Code for this post can be found in this notebook.

. . .

Motivation

When dealing with real problems and real data we often deal with high dimensional data that can go up to millions.
While in its original high dimensional structure the data represents itself best sometimes we might need to reduce its dimensionality.
The need to reduce dimensionality is often associated with visualizations (reducing to 2–3 dimensions so we can plot it) but that is not always the case.
Sometimes we might value performance over precision so we could reduce 1,000 dimensional data to 10 dimensions so we can manipulate it faster (eg. calculate distances).
The need to reduce dimensionality at times is real and has many applications.

Before we start, if you had to choose a dimensionality reduction technique for the following cases, which would you choose?

1. Your system measures distance using the cosine similarity, but you need to visualize it to some non-technical board members which

are probably not familiar with cosine similarity at all—how would you do that?

2. You have the need to compress the data to as little dimensions as you can and the constraint you were given is to preserve approx. 80% of the data, how would you go about that?

3. You have a database of some kind of data that has been collected through a lot of time, and data (of similar type) keeps coming in from time to time.
   You need to reduce the data you have and any new data as it comes, which method would you choose?

My hope in this post to help you understand dimensionality reduction better, so you would feel comfortable with questions similar to those.

Lets start with PCA.

. . .

PCA

PCA (**P**rincipal **C**omponent **A**nalysis) is probably the oldest trick in the book.
PCA is well studied and there are numerous ways to get to the same solution, we will talk about two of them here, Eigen decomposition and Singular Value Decomposition (SVD) and then we will implement the SVD way in TensorFlow.

From now on, X will be our data matrix, of shape (n, p) where n is the number of examples, and p are the dimensions.
So given X, both methods will try to find, in their own way, a way to manipulate and decompose X in a manner that later on we could multiply the decomposed results to represent maximum information in less dimensions. I know I know, sounds horrible but I will spare you most of the math but keep the parts that contribute to the understanding of the method pros and cons.

So Eigen decomposition and SVD are both ways to decompose matrices, lets see how they help us in PCA and how they are connected. Take a glance at the flow chart below and I will explain right after.

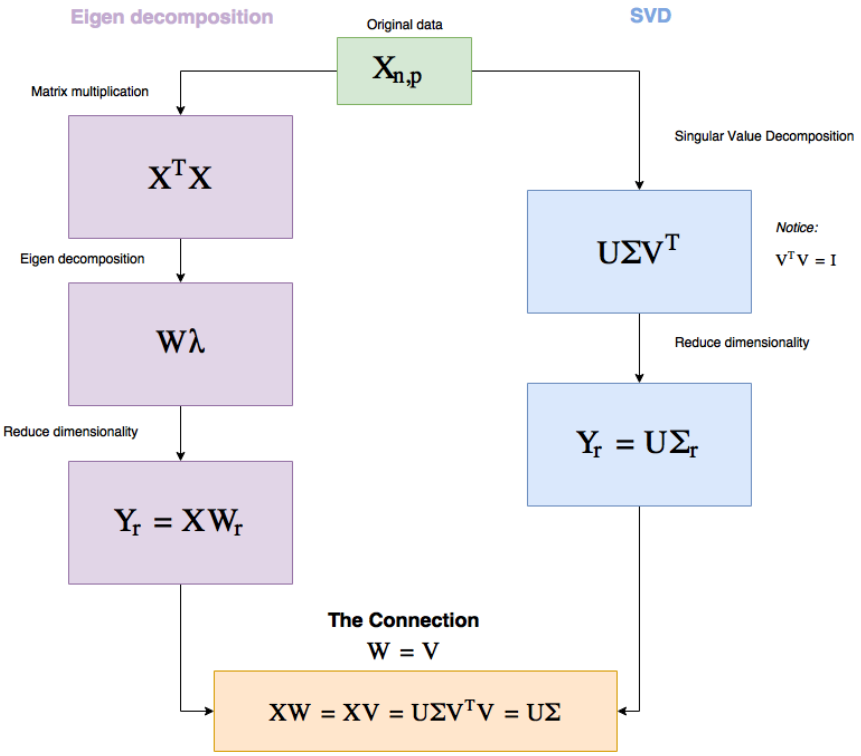

Figure 1 PCA workflow

So why should you care about this? Well there is something very fundamental about the two procedures that tells us a lot about PCA. As you can see both methods are pure linear algebra, that basically tells us that using PCA is looking at the real data, from a different angle—this is unique to PCA since the other methods start with random representation of lower dimensional data and try to get it to behave like the high dimensional data.
Some other notable things are that all operations are linear and with SVD are super-super fast.
Also given the same data PCA will always give the same answer (which is not true about the other two methods).

Notice how in SVD we choose the r (r is the number of dimensions we want to reduce to) left most values of Σ to lower dimensionality?
Well there is something special about Σ .
Σ is a diagonal matrix, there are p (number of dimensions) diagonal values (called singular values) and their magnitude indicates how significant they are to preserving the information.
So we can choose to reduce dimensionality, to the number of dimensions that will preserve approx. given amount of percentage of the data and I will demonstrate that in the code (e.g. gives us the ability to reduce dimensionality with a constraint of losing a max of 15% of the data).

As you will see, coding this in TensorFlow is pretty simple—what we are are going to code is a class that has `fit` method and a `reduce` method which we will supply the dimensions to.

CODE (PCA)

Lets see how the `fit` method looks like, given `self.X` contains the data and `self.dtype=tf.float32`

```
def fit(self):
    self.graph = tf.Graph()
    with self.graph.as_default():
        self.X = tf.placeholder(self.dtype,
shape=self.data.shape)

        # Perform SVD
        singular_values, u, _ = tf.svd(self.X)

        # Create sigma matrix
        sigma = tf.diag(singular_values)

    with tf.Session(graph=self.graph) as session:
        self.u, self.singular_values, self.sigma =
session.run([u, singular_values, sigma],

feed_dict={self.X: self.data})
```

So the goal of `fit` is to create our Σ and U for later use.
We'll start with the line `tf.svd` which gives us the singular values, which are the diagonal values of what was denoted as Σ in Figure 1, and the matrices U and V.
Then `tf.diag` is TensorFlow's way of converting a 1D vector, to a diagonal matrix, which in our case will result in Σ.
At the end of the `fit` call we will have the singular values, Σ and U.

Now lets lets implement `reduce` .

```
def reduce(self, n_dimensions=None, keep_info=None):
    if keep_info:
        # Normalize singular values
        normalized_singular_values =
self.singular_values / sum(self.singular_values)

        # Create the aggregated ladder of kept
information per dimension
        ladder = np.cumsum(normalized_singular_values)
```

```
        # Get the first index which is above the given
information threshold
        index = next(idx for idx, value in
enumerate(ladder) if value >= keep_info) + 1
        n_dimensions = index

    with self.graph.as_default():
        # Cut out the relevant part from sigma
        sigma = tf.slice(self.sigma, [0, 0],
[self.data.shape[1], n_dimensions])

        # PCA
        pca = tf.matmul(self.u, sigma)

    with tf.Session(graph=self.graph) as session:
        return session.run(pca, feed_dict={self.X:
self.data})
```

So as you can see `reduce` gets either `keep_info` or `n_dimensions` (I didn't implement the input check where ***only one must be supplied***). If we supply `n_dimensions` it will simply reduce to that number, but if we supply `keep_info` which should be a float between 0 and 1, we will preserve that much information from the original data (0.9—preserve 90% of the data).
In the first 'if', we normalize and check how many singular values are needed, basically figuring out `n_dimensions` out of `keep_info`.

In the graph, we just slice the Σ (sigma) matrix for as much data as we need and perform the matrix multiplication.

So lets try it out on the iris dataset, which is (150, 4) dataset of 3 species of iris flowers.

```
from sklearn import datasets
import matplotlib.pyplot as plt
import seaborn as sns

tf_pca = TF_PCA(iris_dataset.data,
iris_dataset.target)
tf_pca.fit()
pca = tf_pca.reduce(keep_info=0.9)  # Results in 2
dimensions

color_mapping = {0: sns.xkcd_rgb['bright purple'], 1:
sns.xkcd_rgb['lime'], 2: sns.xkcd_rgb['ochre']}
colors = list(map(lambda x: color_mapping[x],
tf_pca.target))

plt.scatter(pca[:, 0], pca[:, 1], c=colors)
```

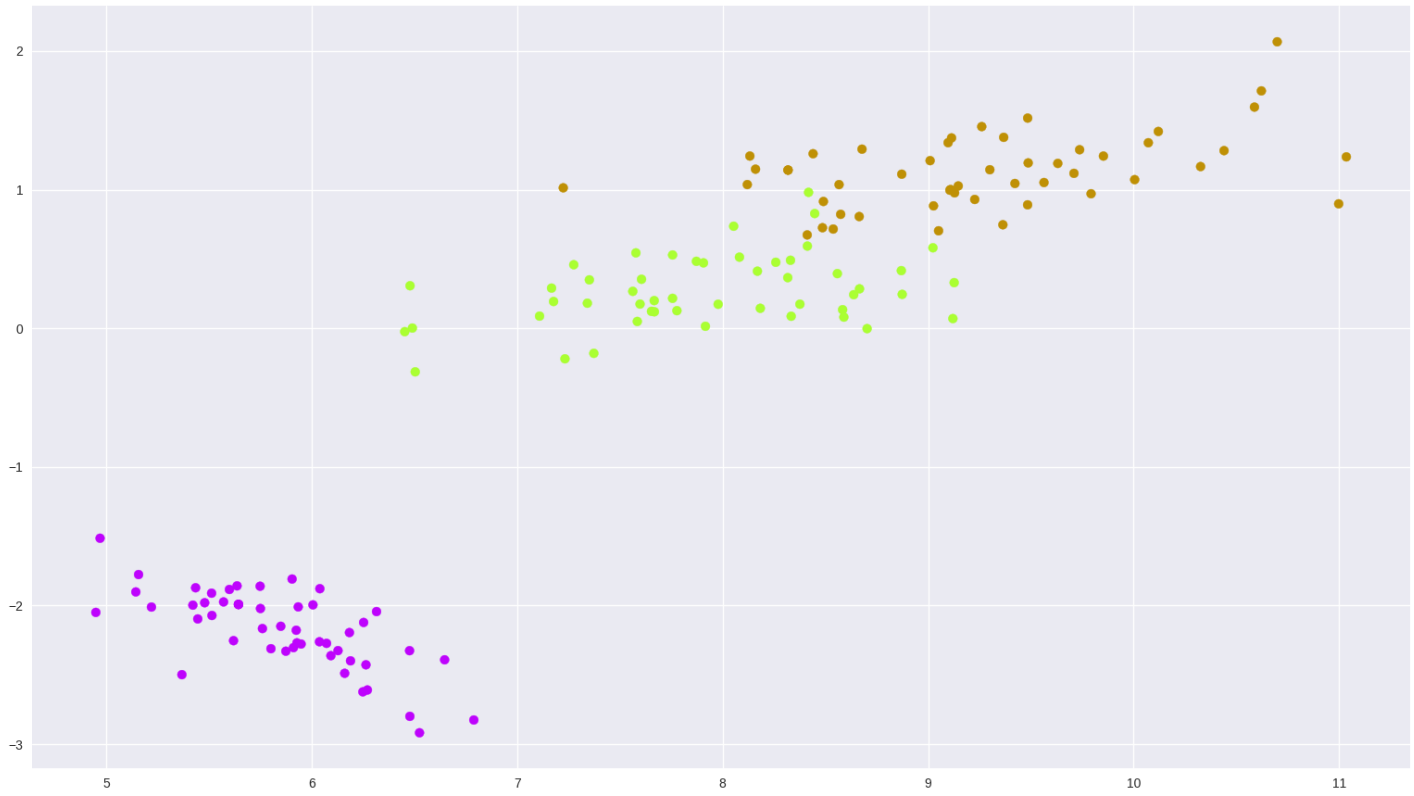

Figure 2 Iris dataset PCA 2 dimensional plot

Not so bad huh?

. . .

t-SNE

t-SNE is a relatively (to PCA) new method, originated in 2008 (original paper link).
It is also more complicated to understand than PCA, so bear with me.
Our notation for t-SNE will be as follows, X will be the original data, P will be a matrix that holds affinities (~distances) between points in X in the high (original) dimensional space, and Q will be the matrix that holds affinities between data points the low dimensional space. If we have n data samples, both Q and P will be n by n matrices (distance from any point to any point including itself).
Now t-SNE has its "special ways" (which we will get to shortly) to measure distances between things, a certain way to measure distance between data points in the high dimensional space, another way for data points in the low dimensional space and a third way for measuring the distance between P and Q.
Taken from the original paper, the similarity between one point x_j to

another point x_i is given by "*p_j|i, that x_i would pick x_j as its neighbor if neighbors were picked in proportion to their probability density under a Gaussian centered at x_*i".
"Whaaat?" don't worry about it, as I said, t-SNE has its ways of measuring distance so we will take a look at the formulas for measuring distances (affinities) and pick out the insights we need from them to understand t-SNE's behavior.

High level speaking, this is how the algorithm works (notice that unlike PCA, it is an iterative algorithm).

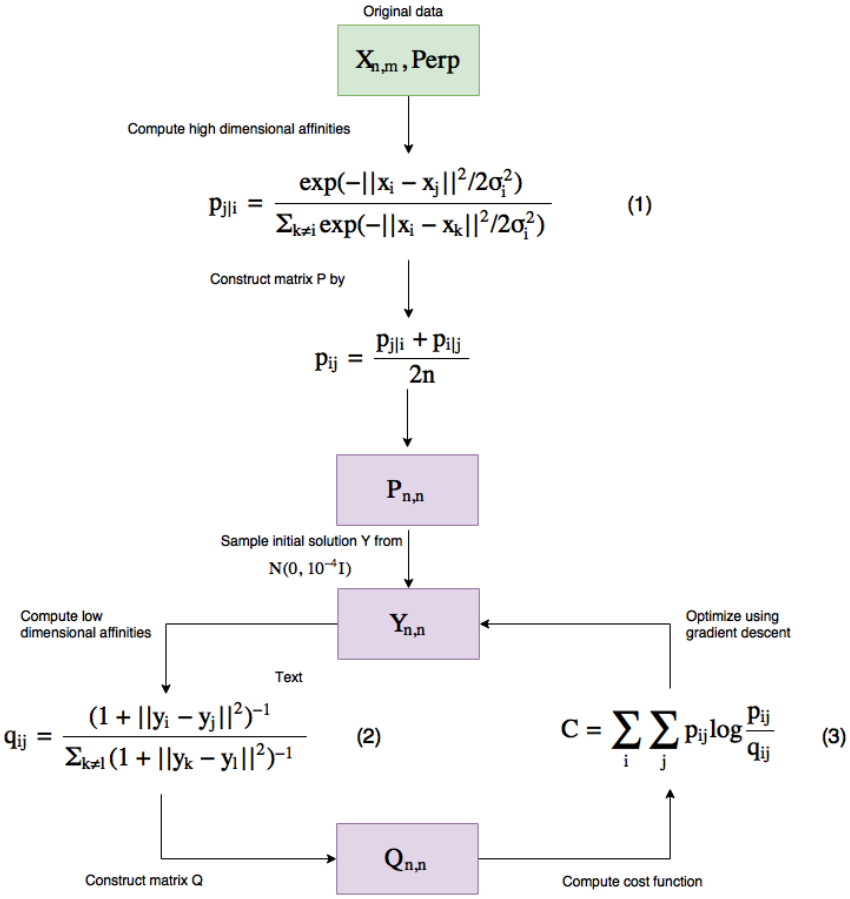

Figure 3 t-SNE workflow

Lets go over this step by step.
The algorithm accepts two inputs, one is the data itself, and the other is called the perplexity (Perp).
Perplexity simply put is how you want to balance the focus between local (close points) and global structure of your data in the optimization process— the article suggests to keep this between 5 and 50.
Higher perplexity means a data point will consider more points as its close neighbors and lower means less.
Perplexity really affects how your visualizations will come up and be

careful with it because it can create misleading phenomenons in the visualized low dimensional data—I strongly suggest reading this great post about how to use t-SNE properly which covers the effects of different perplexities.

Where does this perplexity comes in place? It is the used to figure out σ_i in equation (1) and since they have a monotonic connection it is found by binary search.
So σ_i is basically figured out for us differently, using the perplexity we supply to the algorithm.

Lets see what the equations tells us about t-SNE.
A thing to know before we explore equations (1) and (2) is that p_ii is set to 0 and so does q_ii (even though the equations will not output zero if we apply them on two similar points, this is just a given).
So looking at equations (1) and (2) I want you to notice, that if two points are close (in the high dimensional representation) the numerators will yield a value around 1 while if they are very far apart we would get an infinitesimal—this will help us understand the cost function later.

Already now we can see a couple of things about t-SNE.
One is that interpreting distance in t-SNE plots can be problematic, because of the way the affinities equations are built.
This means that distance between clusters and cluster sizes can be misleading and will be affected by the chosen perplexity too (again I will refer you to the great article you can find in the paragraph above to see visualizations of these phenomenons).
Second thing is notice how in equation (1) we basically compute the euclidean distance between points? There is something very powerful in that, we can switch that distance measure with any distance measure of our liking, cosine distance, Manhattan distance or any kind of measurement you want (as long as it keeps the space metric) and keep the low dimensional affinities the same—this will result in plotting complex distances, in an euclidean way.
For example, if you are a CTO and you have some data that you measure its distance by the cosine similarity and your CEO want you to present some kind of plot representing the data, I'm not so sure you'll have the time to explain the board what is cosine similarity and how to interpret clusters, you can simply plot cosine similarity clusters, as euclidean distance clusters using t-SNE—and that's pretty awesome I'd say.
In code, you can achieve this in `scikit-learn` by supplying a distance matrix to the `TSNE` method.

OK so now that we know that p_ij/q_ij value is bigger when x_i and x_j are close, and very small when they are large.
Lets see how does that affect our cost function (which is called the Kullback–Leibler divergence) by plotting it and examining equation (3) without the summation part.

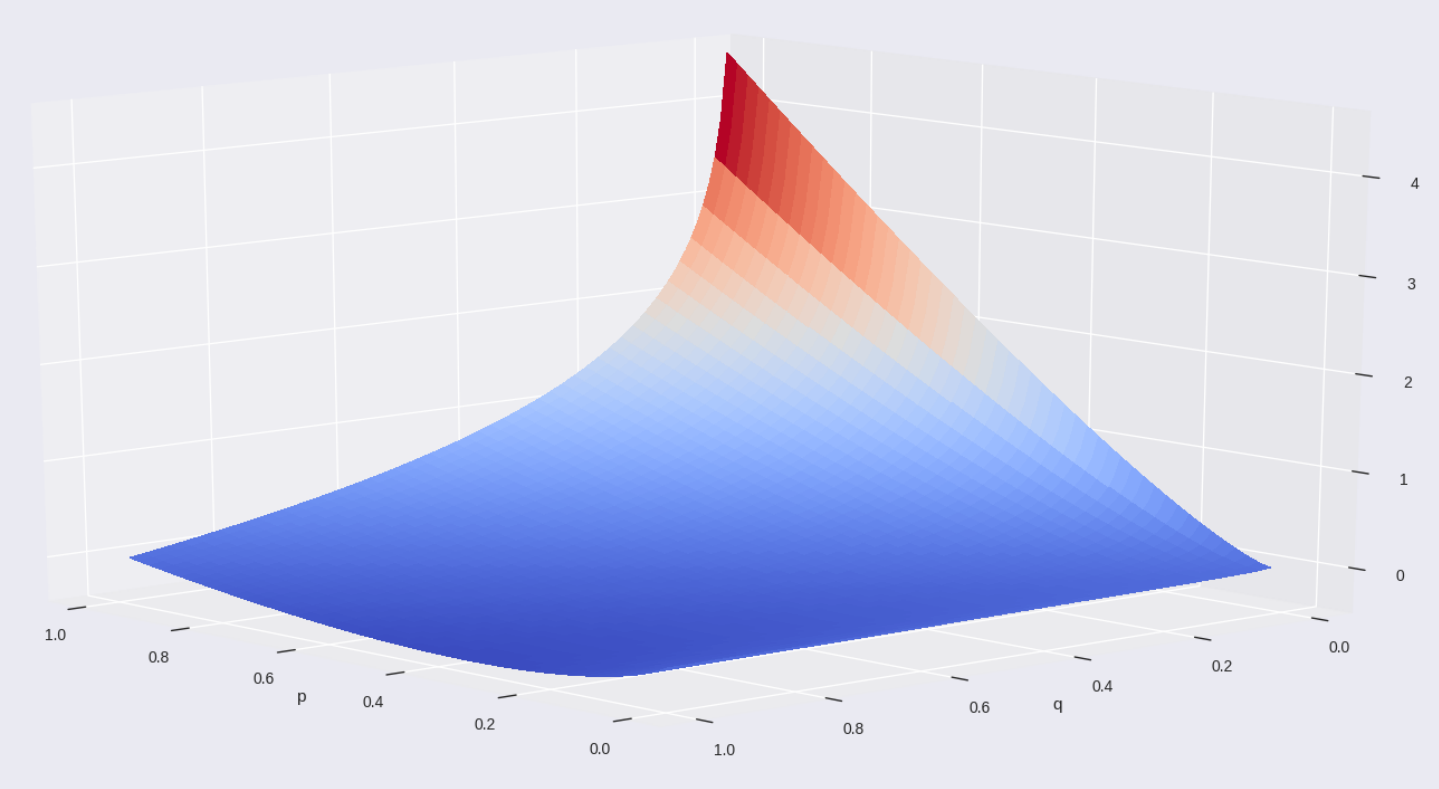

Figure 4 t-SNE cost function without the summation part

Its pretty hard to catch, but I did put the axis names there.
So as you can see, the cost function is asymmetric.
It yields a great cost to points that are nearby in the high dimensional space (p axis) but are represented by far away points in the low dimensional space while a smaller cost for far apart points in the high dimensional space represented by near points in the low dimensional space.
This indicates even more the problem of distance interpret ability in t-SNE plots.

Lets t-SNE the iris dataset and see what happens with different perplexities

```
model = TSNE(learning_rate=100, n_components=2,
random_state=0, perplexity=5)
tsne5 = model.fit_transform(iris_dataset.data)

model = TSNE(learning_rate=100, n_components=2,
random_state=0, perplexity=30)
tsne30 = model.fit_transform(iris_dataset.data)

model = TSNE(learning_rate=100, n_components=2,
random_state=0, perplexity=50)
tsne50 = model.fit_transform(iris_dataset.data)

plt.figure(1)
plt.subplot(311)
plt.scatter(tsne5[:, 0], tsne5[:, 1], c=colors)

plt.subplot(312)
plt.scatter(tsne30[:, 0], tsne30[:, 1], c=colors)

plt.subplot(313)
plt.scatter(tsne50[:, 0], tsne50[:, 1], c=colors)

plt.show()
```

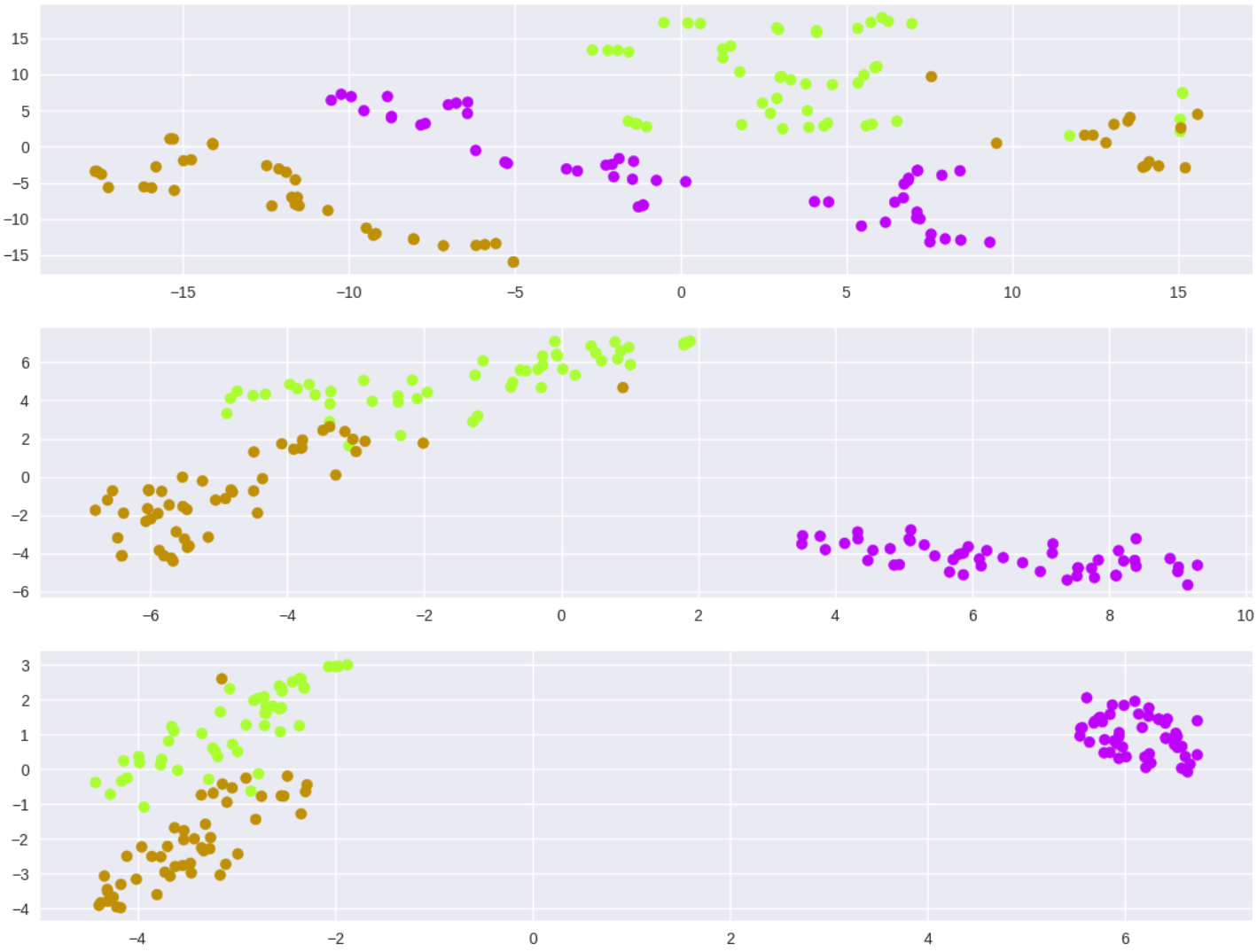

Figure 5 t-SNE on iris dataset, different perplexities

As we understood from the math, you can see that given a good perplexity the data does cluster, but notice the sensibility to the hyperparameters (I couldn't find clusters without supplying learning rate to the gradient descent).
Before we move on I want to say that t-SNE is a very powerful method if you apply it correctly and don't take what you've learned to the negative side, just be aware of how to use it.

Next are Auto Encoders.

. . .

Auto Encoders

While PCA and t-SNE are methods, Auto Encoders are a family of methods.
Auto Encoders are neural networks where the network aims to predict the input (the output is trained to be as similar as possible to the input) by using less hidden nodes (on the end of the encoder) than input nodes by encoding as much information as it can to the hidden nodes.
A basic auto encoder for our 4 dimensional iris dataset would look like Figure 6, where the lines connecting between the input layer to the hidden layer are called the “encoder” and the lines between the hidden layer and the output layer the “decoder”.

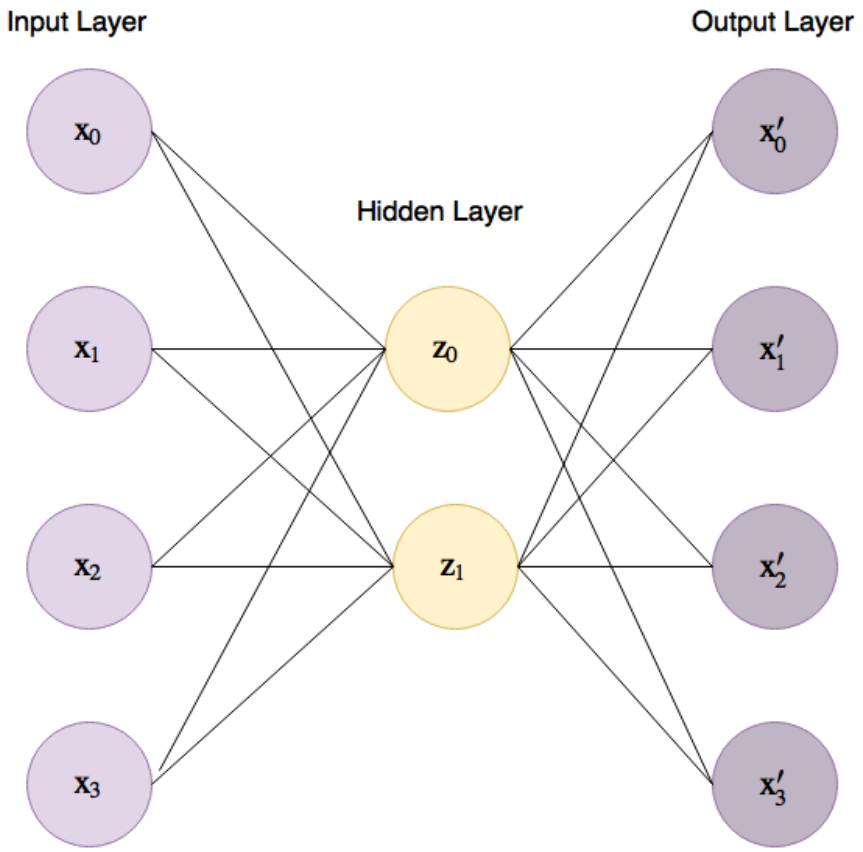

Figure 6 Basic auto encoder for the iris dataset

So why are Auto Encoders are a family? Well because the only constraint we have is that the input and output layer will be of the same dimension, inside we can create any architecture we want to be able to encode best our high dimensional data.

Auto Encoders starts with some random low dimensional representation (z) and will gradient descent towards their solution by changing the weights that connect the input layer to the hidden layer, and the hidden layer to the output layer.

By now we can already learn something important about Auto Encoders, because we control the inside of the network, we can engineer encoders that will be able to pick very complex relationships between features.
Another great plus in Auto Encoders, is that since by the end of the training we have the weights that lead to the hidden layer, we can train on certain input, and if later on we come across another data point we can reduce its dimensionality using those weights without re-training —but be careful with that, this will only work if the data point is somewhat similar to the data we trained on.

To explore the math of Auto Encoder could be simple in this case but not quite useful, since the math will be different for every architecture and cost function we will choose.
But if we take a moment and think about the way the weights of the Auto Encoder will be optimized we understand the the cost function we define has a very important role.
Since the Auto Encoder will use the cost function to determine how good are its predictions we can use that power to emphasize what we want to.
Whether we want the euclidean distance or other measurements, we can reflect them on the encoded data through the cost function, using different distance methods, using asymmetric functions and what not.
More power lies in the fact that as this is a neural network essentially, we can even weight classes and samples as we train to give more significance to certain phenomenons in the data.
This gives us great flexibility in the way we compress our data.

Auto Encoders are very powerful and have shown some great results in comparison to other methods in some cases (just Google “PCA vs Auto Encoders”) so they are definitely a valid approach.

Lets TensorFlow a basic Auto Encoder for the iris data set and plot it

CODE (Auto Encoder)

Again, we’ll split into `fit` and `reduce`

```
def fit(self, n_dimensions):
    graph = tf.Graph()
    with graph.as_default():

        # Input variable
        X = tf.placeholder(self.dtype, shape=(None,
self.features.shape[1]))
```

```
        # Network variables
        encoder_weights =
tf.Variable(tf.random_normal(shape=
(self.features.shape[1], n_dimensions)))
        encoder_bias = tf.Variable(tf.zeros(shape=
[n_dimensions]))

        decoder_weights =
tf.Variable(tf.random_normal(shape=(n_dimensions,
self.features.shape[1])))
        decoder_bias = tf.Variable(tf.zeros(shape=
[self.features.shape[1]]))

        # Encoder part
        encoding = tf.nn.sigmoid(tf.add(tf.matmul(X,
encoder_weights), encoder_bias))

        # Decoder part
        predicted_x =
tf.nn.sigmoid(tf.add(tf.matmul(encoding,
decoder_weights), decoder_bias))

        # Define the cost function and optimizer to
minimize squared error
        cost =
tf.reduce_mean(tf.pow(tf.subtract(predicted_x, X), 2))
        optimizer =
tf.train.AdamOptimizer().minimize(cost)

    with tf.Session(graph=graph) as session:
        # Initialize global variables
        session.run(tf.global_variables_initializer())

        for batch_x in batch_generator(self.features):
            self.encoder['weights'],
self.encoder['bias'], _ =
session.run([encoder_weights, encoder_bias,
optimizer],

feed_dict={X: batch_x})
```

Nothing special in here, code is pretty self explanatory and we save our encoders weights in biases, so we could reduce the data in the `reduce` method which is here next.

```
def reduce(self):
    return np.add(np.matmul(self.features,
self.encoder['weights']), self.encoder['bias'])
```

Boom, that simple :)

Lets see how did it do (batch size 50, 1000 epochs)

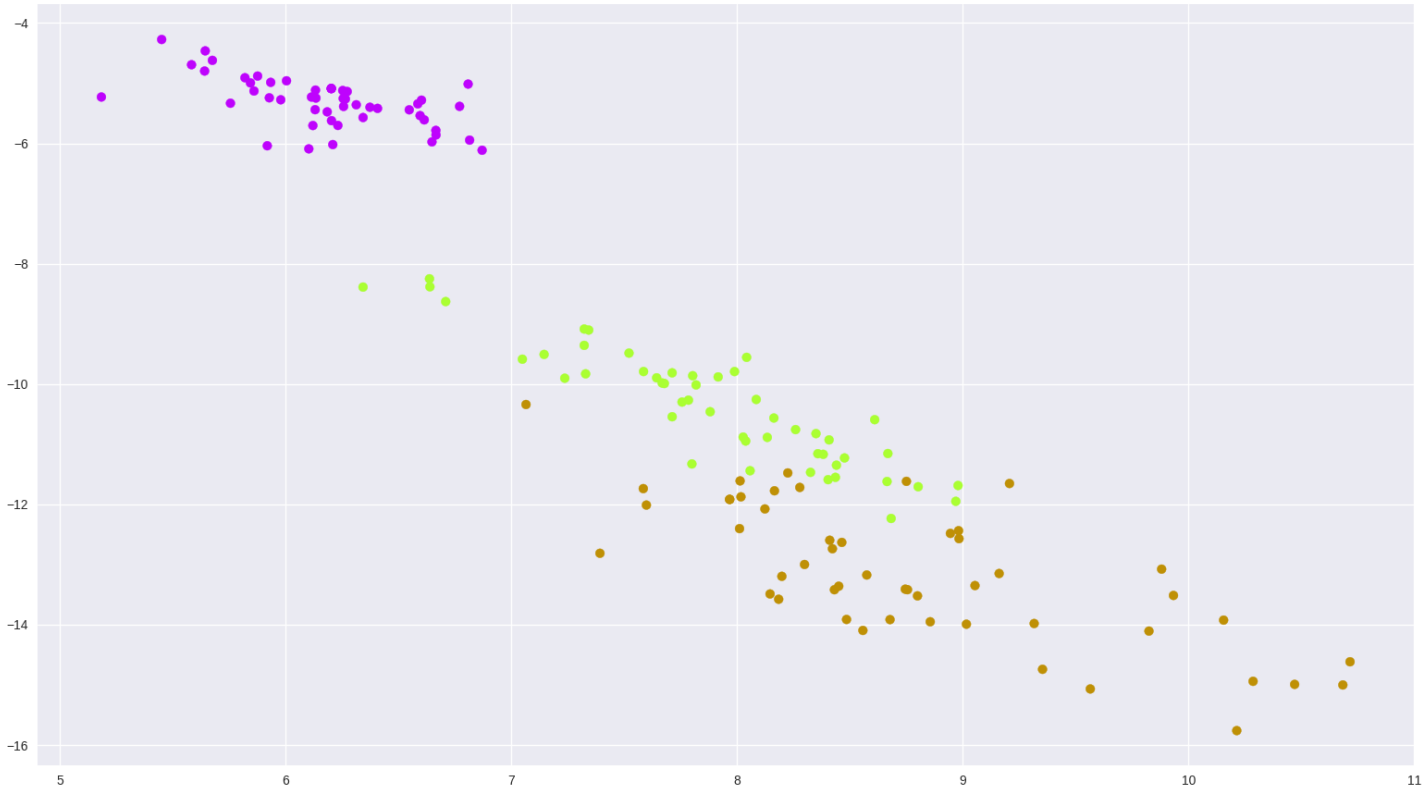

Figure 7 Simple Auto Encoder output on iris dataset

We could continue to play with the batch size, number of epochs and different optimizers even without changing the architecture and we would get varying results—this is what came just off the bat.
Notice I just chose some arbitrary values for the hyperparameters, in a real scenario we would measure how well we are doing by cross validation or test data and find the best setting.

Final Words

Posts like this usually end with some kind of comparison chart, pros and cons etc.
But that is the exact opposite of what I was trying to achieve.
My goal was to expose the intimate parts of the methods so the reader would be able to figure out and understand positives and negatives of each one.
I hope you enjoyed the reading and have learnt something new.

Scroll up to the beginning of the post, to those three questions, feel any more comfortable now with them?