1. **Introduction**

The topic of the discussion is Manifold Learning, or, put more simply, non-linear methods of dimensionality reduction. When we have data that has a lot of dimensions, we could face certain problems during analysis: high computational cost and the consequences of the infamous Curse of dimensionality are the first things that come to mind.
Manifold learning attempts to address these problems by providing us with data with fewer dimensions. This is an alternative to feature selection: instead of trying to choose the variables that carry the most meaning and disposing of the rest, Manifold Learning algorithms attempt to sort of “compress” all of the variables into a lower-dimensional representation. As a result, we, first of all, lose some information - that much is inevitable and must be always remembered. But second - we get an easier-to-work-with dataset containing an arbitrarily small number of features.
My partner here and me attempted to compare the methods based on the construction of the neighborhood graphs and Laplacian matrices with a Neural Network based method - an Autoencoder. Our work follows in the steps of James McQueen and Co called “megaman: Manifold Learning with millions of points”.
The rest of our presentation is organized as follows:

a) First, I will briefly remind what an Autoencoder is
b) Second, I will talk about my own implementations of AE.
c) Third, we will see how my AEs performed in terms of speed and quality following the tests carried out in the article
d) Finally, my partner will give an overview of performances for other manifold learning methods.

2. Autoencoders.
AE is a type of NN that has the goal to learn a representation of the given data (which would, typically, be in lower dimension than the original) in an unsupervised fashion. AE can be split into three parts - encoder, latent space and decoder.

3. I implemented 3 AEs which basically can be described as a vanilla model where both encoder and decoder are represented by just one fully connected layer each, an upgraded version of the same model where I added batch normalization and activation functions not prone to vanishing gradients and, finally, a more complex CNN model
The fact that we have three models of increasing complexity will allow for a demosntration of some properties of neural nets. Also, they gave different results (in terms of speed and quality).

As suggested in the paper, first I tested my AEs on noisy Swiss Roll dataset. All three performed reasonably well, clearly beating scikit-learn’s implementation of Spectral Embedding both in speed and quality. Here are the examples produced from a 15-dimensional noisy Swiss Roll. (show plots). This was a “sanity check”, just a way to see if AE was doing something useful at all.

I then repeated the tests suggested in paper - first, fixed number of dimensions D=100 and varied the number of observations in between 10000 and 1000000. I used my second model

(middle complexity). The time it took to train was affected by the mechanism of crude early stopping I implemented. Here is the dependency of training time on number of observations (show plots). After this test, I varied the number of dimensions while keeping N=50000. Here is the plot
Megaman was faster than NN, but not by much. I think a network would scale better - we could make use of transfer learning and just update our network instead of retraining from scratch, whereas the neighborhood graph and Laplacian would have to be redone from zero.

Just to make sure that my AEs still are doing good work, here are some encodings produced after training (show plots)

Finally, word2vec. Here, the results look like this (show).
This does not really resemble the plots from article of McQueen. Why is that? I think, the reason for that lies in the principles of work of an AE. First, we compress our data to latent space. Then, we recover it. The loss function of my AE is the L1 loss between input and output.
So, basically, we want to be able to compress 300 dimensions into 2 and back. The qualoty of AE depends directly on its ability to do that.
However, think about it - we delete 298 dimensions out of 2. It stands to reason that we lose a lot of information. Therefore, decoder is unable to reconstruct well and thus the quality of AE degrades.
The reason it worked well with Swiss Roll is explainable in the same way - there, we had just 3 meaningful dimensions and then arbitrary N dimensions of noise. THAT is compressable into 3 dimensions very well. In word2vec, we cannot same the same - the majority of these 300 dimensions matters. Hence the difference in quality.

So, to sum up:

1. AEs work well when the compression is reasonable (most of information is preserved)
2. AEs scale better - they are fairly quick to train and due to the nature of NNs are able to use of transfer learning
3. When we want to compress data extremely tightly - from 300 to 2, for example - neighborhood graph methods are to be preferred.
4. Complex networks require more data to train - due to higher number of weights.
5. No free lunch - there is no one good NN that will work for every dataset. Improvise. Adapt. Overcome.
6. Plotly is awesome, use it instead of matplotlib when you can.