Mar 1-Unsupervised Learning/refs15.bib

COMP 642 — Machine Learning March 1, 2022

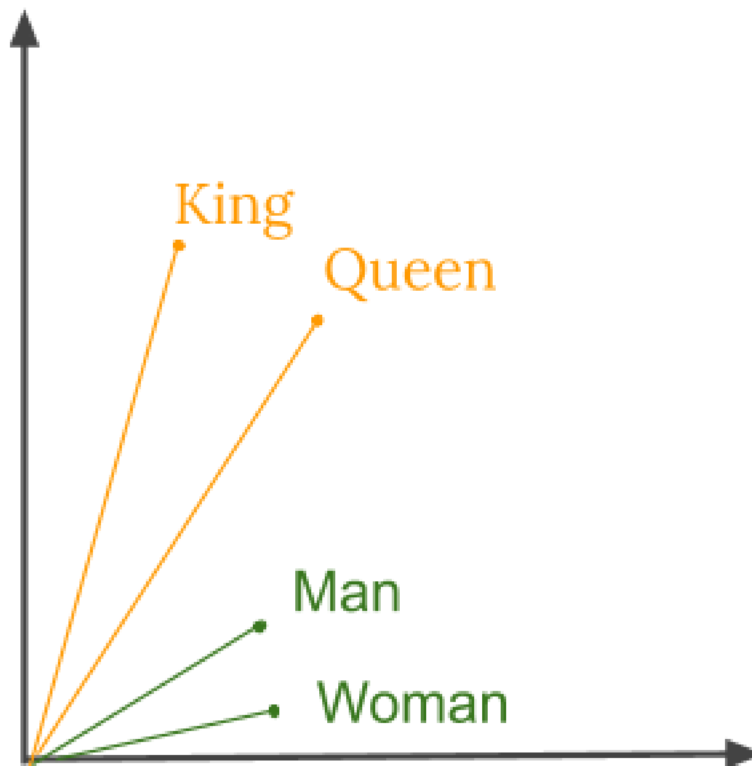
Lecture 15: Unsupervised Learning

Lecturer: Anshumali Shrivastava
Scribe By: Bowen Deng, Xilin Song, Zijian Zhou, Angel Chen

Disclaimer: *These lecture notes are intended to develop the thought process and intuition in machine learning. The materials are not thoroughly reviewed and can contain errors.*

1 Word Embeddings and Word2Vec

Word embedding is one of the most popular representations of document vocabulary. It is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc. They have learned representations of text in an n-dimensional space where words that have the same meaning have a similar representation. Meaning that two similar words are represented by almost similar vectors that are very closely placed in a vector space. In short, word embeddings are vector representations of a particular word. Therefore, by using word embeddings, all individual words are represented as real-valued vectors in a predefined vector space, as shown in the image below. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network.

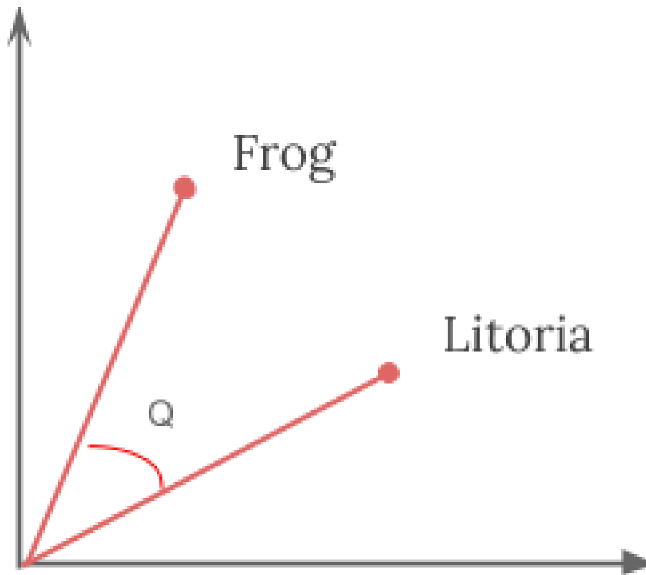


1.1 Word embedding vs One-hot encoding

Consider a simple example, assuming there are two sentences: "You can scale your business." and "You can grow your business." two sentences have the same meaning. If we consider a vocabulary considering these two sentences, it will constitute of these words: You, can, scale, grow, your, business.

A one-hot encoding of these words would create a vector of length 6. The encodings for each of

words are really close. For example, In the image below, the cosine of the angle between the vector representation of words 'Frog' and 'Litoria' should be close to 1. [1]

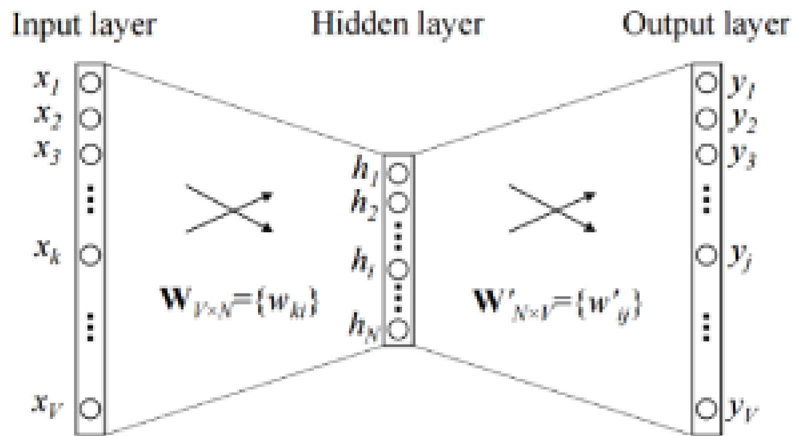


Word2Vec is a method to construct such an embedding. It can be obtained using two methods: Common Bag of Words (CBOW) and Skip Gram.

1.2.1 CBOW Model

CBOW model takes the context of each word as the input and tries to predict the word corresponding to the context.

Consider an example: Have a great day. Let's say, the input to the Neural Network is a single word 'great'. we are trying to predict a target word (day) using a single context input word 'great'.

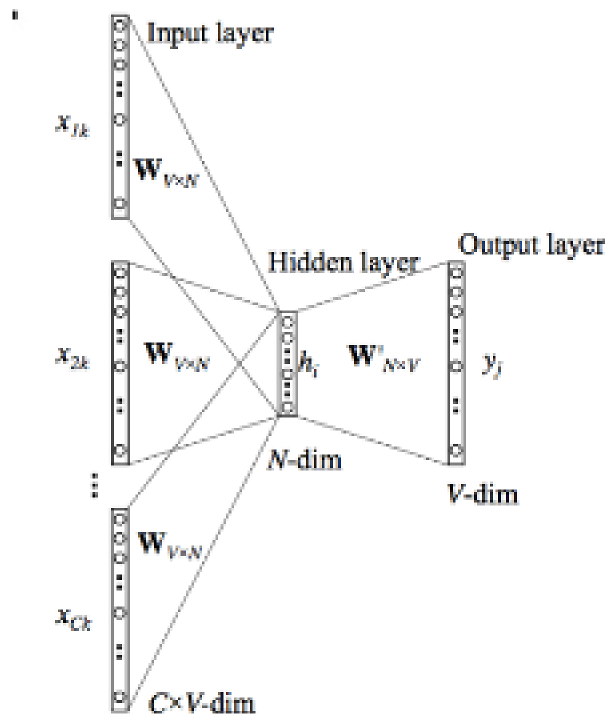


As shown on the figure above, the input or the context word is a one hot encoded vector of size V. The hidden layer contains N neurons, and the output is again a V length vector with

the elements being the softmax values.
Terms:

- Wvn is the weight matrix that maps the input x to the hidden layer (V*N dimensional matrix)
- W'nv is the weight matrix that maps the hidden layer outputs to the final output layer (N*V dimensional matrix)

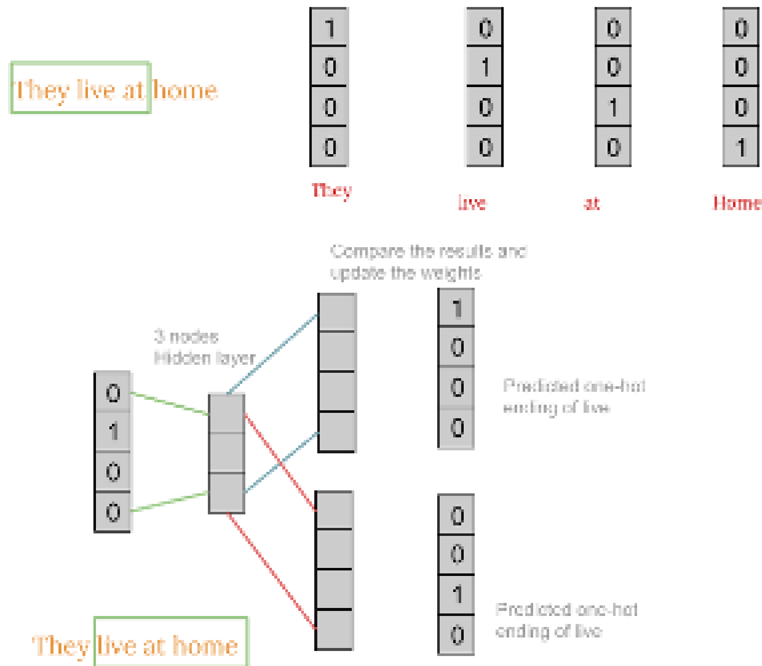
The hidden layer neurons just copy the weighted sum of inputs to the next layer. There is no activation like sigmoid, tanh or ReLU. The only non-linearity is the softmax calculations in the output layer2. Similarly, we can use multiple context words to do the same.



The above model takes C context words. When Wvn is used to calculate hidden layer inputs, we take an average over all these C context word inputs. In this model, we use the target word (whose representation we want to generate) to predict the context and in the process, we produce the representations.

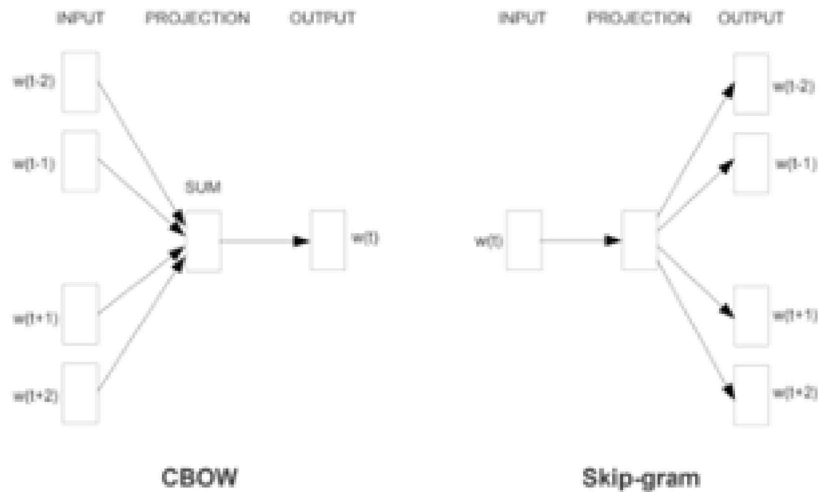
1.2.2 Skip-Gram model

The Skip-gram model usually tries to achieve the reverse of what the CBOW model does. It tries to predict the source context words given a target word. The way of the skip-gram model works is quite similar to the CBOW but there is just a difference in the structure of its neural network and the way the weight matrix is generated as shown in the figure below:



After obtaining the weight matrix, the steps to get word embedding is same as CBOW. [3]

1.2.3 CBOW vs Skip-Gram



Both CBOW and Skip-Gram have their own advantages and disadvantages. Based on the analysis mentioned before, for large corpus with higher dimensions, it turns out that it is better to use skip-gram but is slow to train. Whereas CBOW is better for small corpus and is faster to train. On the other hand, According to Mikolov's work, Skip Gram works well with small amounts of data and is found to represent rare words well. And CBOW is faster and has better representations for more frequent words.

2 Autoencoders

2.1 What is Autoencoder?

An autoencoder is a type of artificial neural network (ANN) used to learn data encodings (compression of the row data) in an unsupervised manner.

The aim of an autoencoder is to learn a lower-dimensional representation (encoding) for a higher-dimensional data. In other words, capturing the most important parts of the input image through the dimensionality reduction to ignore signal noise (Figure 1).

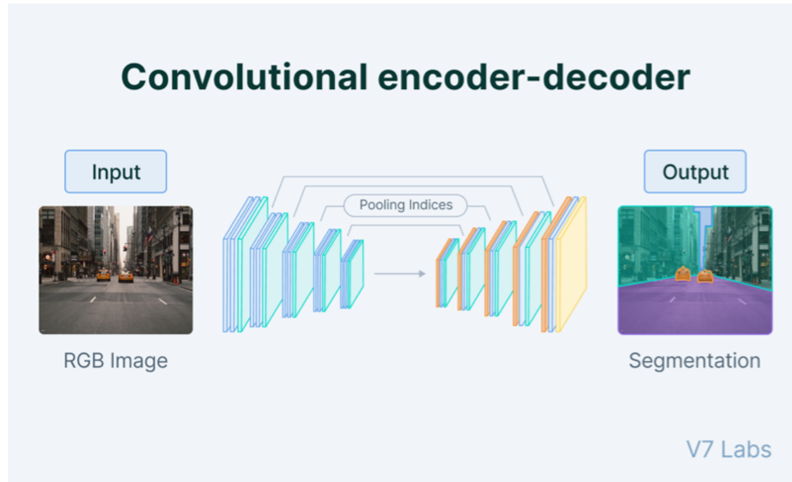


Figure 1: Convolution Autoencoder

2.1.1 The architecture of autoencoders

An autoencoder consists of three components (figure 2 shows the idea of the architecture):

- **Encoder** is a feedforward process that compresses the input data into an encoded representation in a reduced, where the compressed image is the disordered version of the original image.
- **Bottleneck** (a.k.a latent space), contains the knowledge representation of the input that will be fed into the decoder. The bottleneck as a compressed representation of the input prevents the network from memorizing the input and overfitting on the data. The key point to remember is that the smaller the bottleneck, the lower the risk of overfitting. However, the small bottleneck would limit the amount of stored info which may increase the chance of eliminating the important info.
- **Decoder** is a feedforward network which is similar to encoder, instead it helps decompress the knowledge representation and reconstruct the data back to the form of original data.

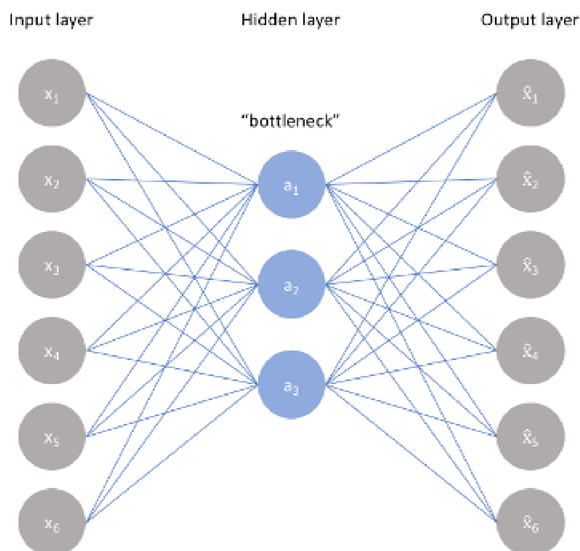


Figure 2: General Autoencoder Architecture

2.2 Types of autoencoders

There are various types of autoencoders, and we will focus on the 5 popular autoencoders:

1. **Undercomplete autoencoders**
   Undercomplete autoencoders are also known as vanilla autoencoder. The idea of undercomplete autoencoder is taking the same input image and tries to predict and capture the most important features present in the data as output. Then, reconstructing the image from the compressed bottleneck region.

   Undercomplete autoencoders have a smaller dimension for the hidden layer compared to the input layer. This helps to obtain important features from the data. It minimizes the loss function by penalizing the g(f(x)) for being different from the input x.

   While trying to compress the input, if we do this with linear layers and squared L2 norm, we use a method like PCA (Principal Component Analysis). Since PCA can only build linear relationships, we cannot support the nonlinear relationship of the undercomplete autoencoder (also known as manifold learning). However, we can reduce the undercomplete autoencoder by removing all non-linear activations, then we can fit the data into the PCA.

2. **Sparse autoencoders**
   Sparse autoencoders are similar to the undercomplete autoencoders in that they use the same image as input and ground truth, but the way of their encoding process is different.

   A sparse autoencoder is an autoencoder whose training criterion involves a sparsity penalty. The loss function is constructed by penalizing activations of hidden layers; therefore, a few nodes are encouraged to activate when a single sample is fed into the network. There are two different ways to construct the sparse penalty: L1 regularization and KL-divergence. Figure 3 shows the idea of the sparse autoencoder.

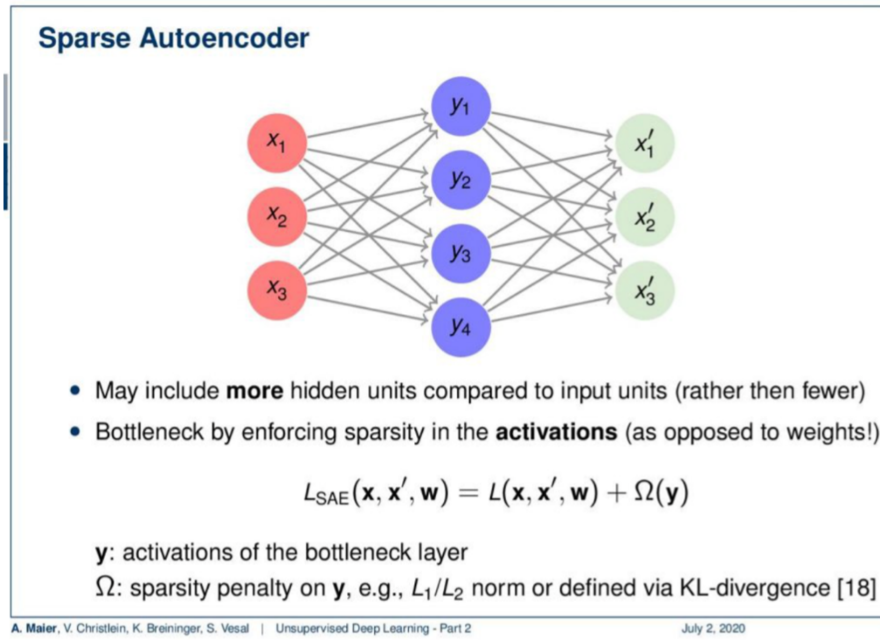


Figure 3: Sparse Autoencoder

3. **Contractive autoencoders**
   The idea of the contractive autoencoders is similar to others' autoencoders. The goal of a contractive autoencoder is to make the encoding proves less sensitive to small variations in its training dataset. This is accomplished by applying a penalty term to the loss function. The regularizer needs to conform to the Frobenius norm of the Jacobian matrix of the encoder activations with respect to the input. The Frobenius norm of the Jacobian matrix for the hidden layer is calculated with respect to input and it is basically the sum of squares of all elements.

   The pros of the contractive autoencoders:

   - A better choice than denoising autoencoder to learn useful feature extraction.
   - The model learns an encoding in which similar inputs have similar encodings. Hence, we're forcing the model to learn how to contract a neighborhood of inputs into a smaller neighborhood of outputs.

4. **Denoising autoencoders**
   This is the first autoencoder that we have discussed so far that does not have the input image as its ground truth.

   Denoising autoencoders takes in a very noisy version of the input image while training to recover the original undistorted input, where the model learns a vector field for mapping the input data towards a lower-dimensiona manifold which describes the natural data to cancel out the added noise. Furthermore, denoising autoencoder works with the help of non-linear dimensionality reduction. The loss function generally used in these types of networks is L2 or L1 loss.

   Figure 4 shows the general idea of the denoising autoencoders.

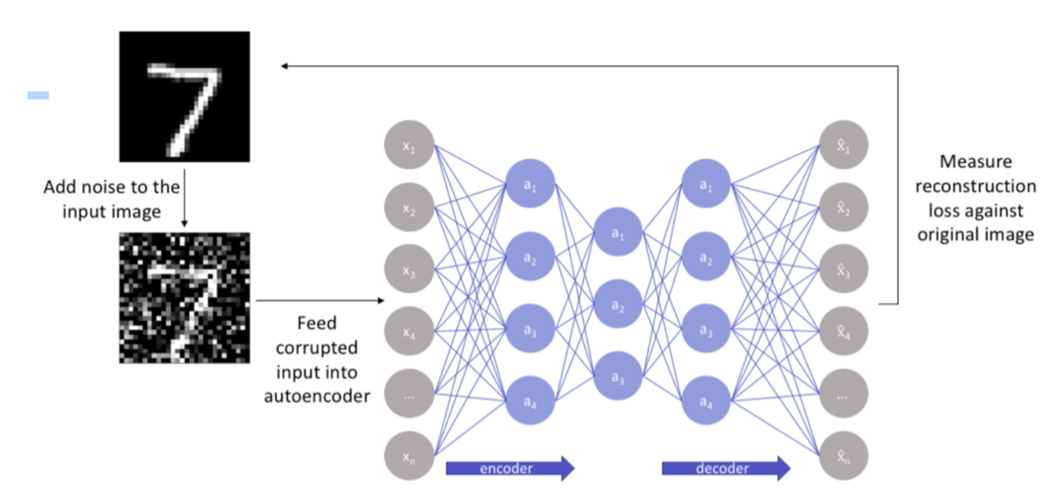


Figure 4: Denoising Autoencoder

The pros of the denoising autoencoder:

- Corrupted input can be done randomly by making some of the input as zero. Remaining nodes copy the input to the noised input.
- Minimizes the loss function between the output node and the corrupted input.

The cons of the denoising autoencoder:

- To train an autoencoder to denoise data, it is necessary to perform preliminary stochastic mapping in order to corrupt the data and use as input.
- Can't develop a mapping which memorizes the training data because our input and target output are no longer the same.

5. **Variational Autoencoders** A variational autoencoder (VAE) provides a probabilistic manner for describing an observation in latent space. In other words, instead of building an encoder which outputs a single value to describe each latent state attribute, VAE formulates the encoder to describe a probability distribution for each latent attribute.

   The classic example in Figure 5 shows how an ideal autoencoder learn from the input.

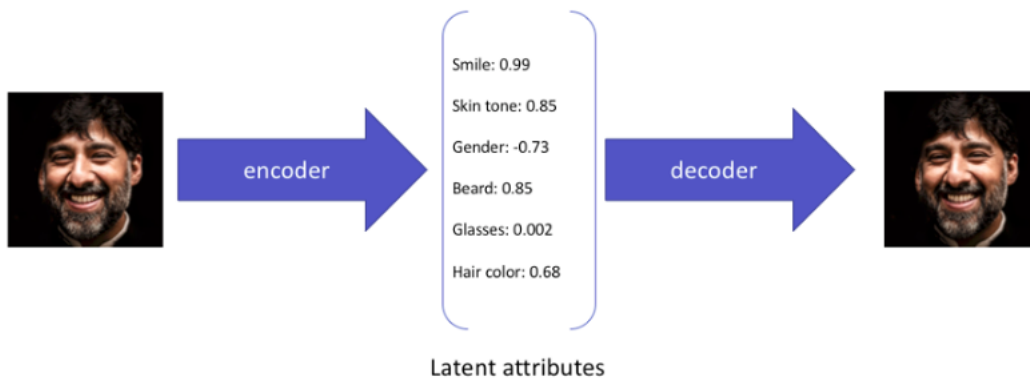


Figure 5: An Example of Ideal Autoencode

As we can see in Figure 5, the input image was described in terms of its latent attributes using a single value to describe each attribute. However, variational autoencoders represent the latent attributes as a probability distribution. By presenting in a probability distribution, the model can be able to format in a continuous latent space, which can be easily sampled and interpolated (as shown in Figure 6).

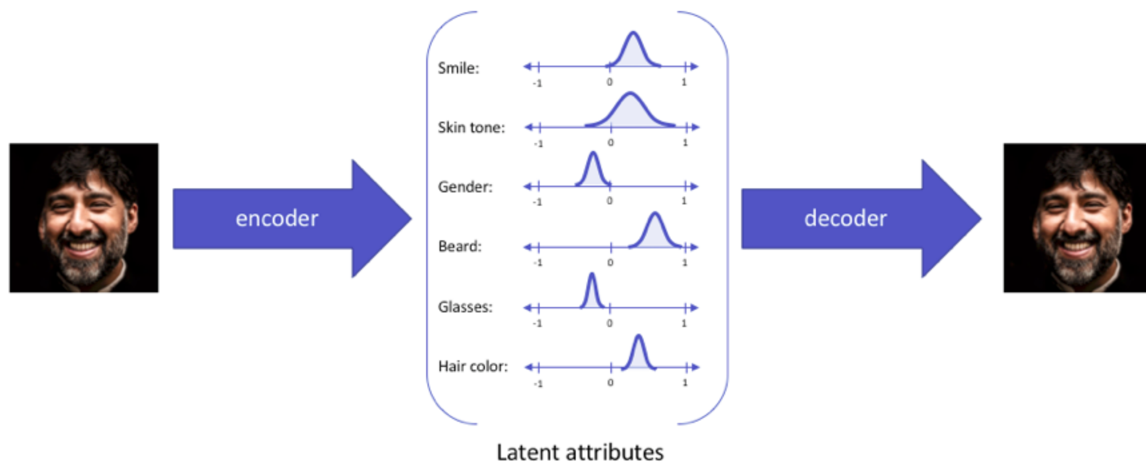


Figure 6: An Example of Variational Autoencoder

Figure 7 further shows an idea of how variational autoencoder works. We expect the decoder model to be able to accurately reconstruct the input for every sampling of the latent distributions. Hence, the values which are nearby to one another in latent space should correspond with a similar reconstruction.

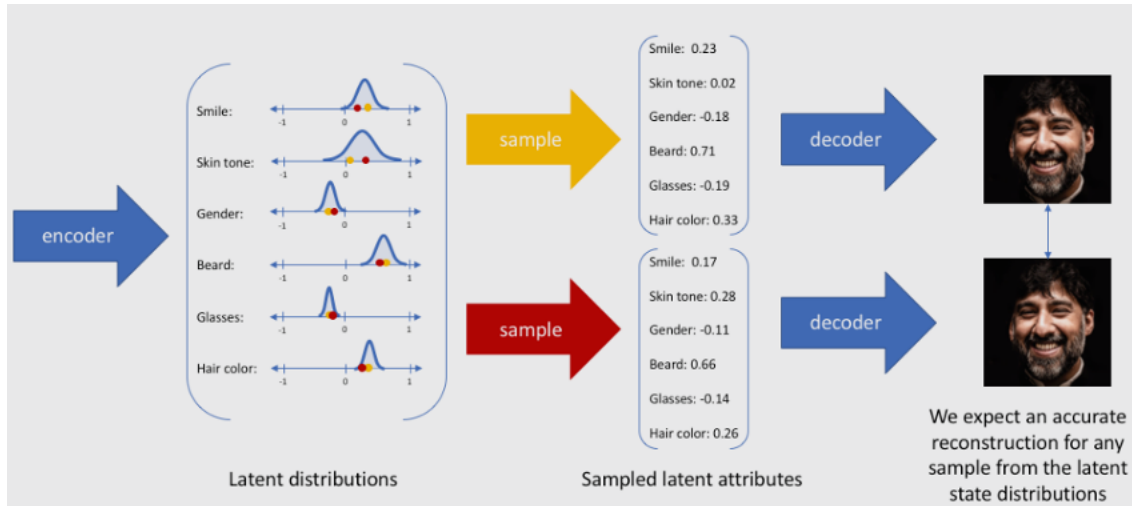


Figure 7: Nearby samples of the latent distributions

The motivation of presenting the latent attributes as a probability are based on the statistical expression, but we will not go into details. In case, if you are interested in, please refer to the citation number [X].

There are still many other types of autoencoders, such as convolutional autoencoder, deep autoencoder, etc.

2.3 Applications of autoencoders

There are some common use cases for the autoencoders:

1. **Dimensionality reduction/Anomaly Detection** uses undercomplete autoencoders.
2. **Image denoising** uses denoising autoencoders.
3. **Generation of image and time series data** uses variational autoencoders.
4. **Face completion** uses denoising autoencoders technique and will be discussed in section 2.4

2.4 Example of Autoencoder Application - Face Completion

2.4.1 Project Overview

The original project was to create a pipeline for photo restoration of portrait images. Shortly after the project started, the researchers focused on the interesting subproblem: facial image completion. Given an image of a face with a rectangular section of the image set to be white, they attempted to fill in the missing pixels.

Figure 8: Original image and masked image

The approach is based on Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks and Enhancing Images Using Deep Convolutional Generative Adversarial Networks (DCGANs). They used a generator, a collection of convolution and deconvolution layers, to reconstruct the original unmasked image.

2.4.2 Dataset

They used the ColabA Dataset, consisting of around 200,000 images including over 10,000 unique identities. And they reserved 1000 images as a test set and 1000 images as a validation set. For each image, they set a randomly sized patch to be white. Each generated patch is at least 10 pixels in width/height and takes up a maximum of 1/3 of the image. The patch's X,Y coordinates are selected randomly.

2.4.3 Architecture

The architecture has two major components: the autoencoder, and the discriminator. A diagram of the architecture is shown below. The autoencoder (left side of diagram) accepts a masked image as an input, and attempts to reconstruct the original unmasked image. The discriminator (right side) is trained to determine whether a given image is a face. The discriminator is run using the output of the autoencoder. The result is used to influence the cost function used to update the autoencoder's weights.

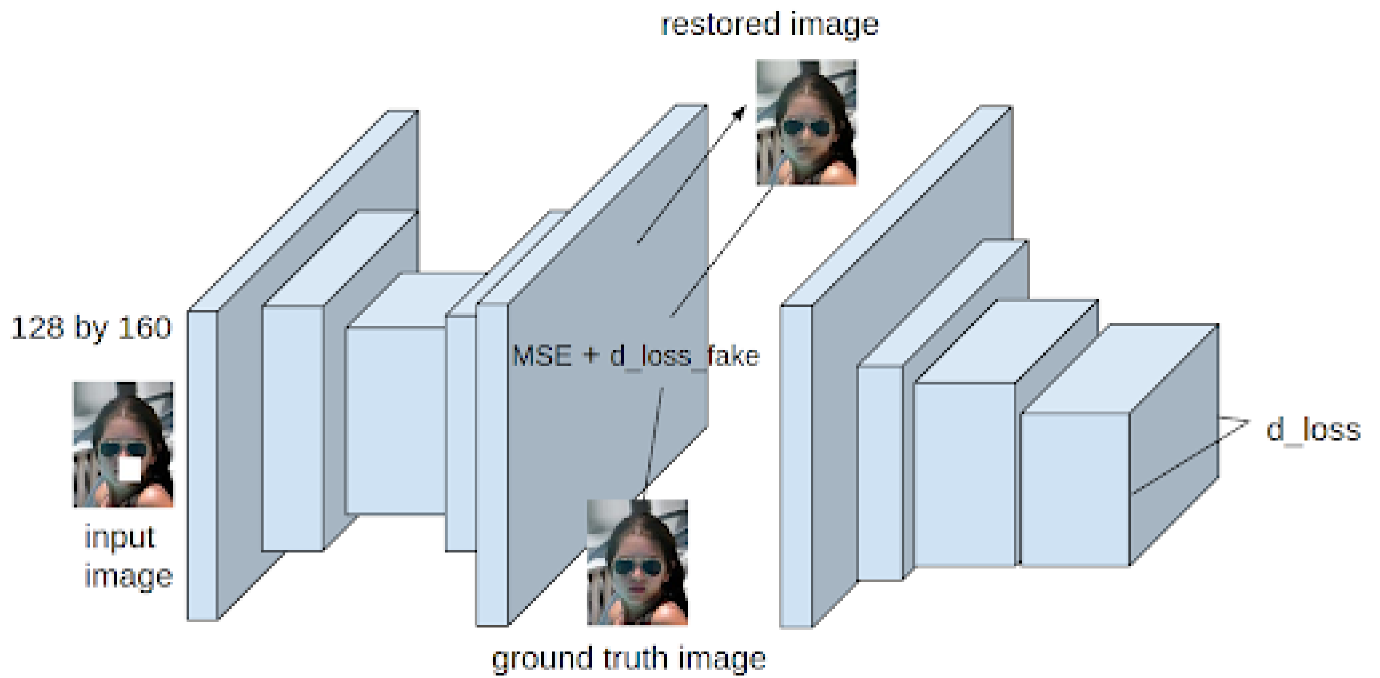


Figure 9

Traditionally, autoencoders have been used to learn a feature representation for some data set. Denoising autoencoders artificially corrupts input data in order to force a more robust representation to be learned. In this case, the image mask is the data corruption. The autoencoder first transforms the input data through a series of 4 convolution layers. At the center layer of the autoencoder, the image is represented by 1024 8*10 feature maps. The input image is then reconstructed by 4 fractionally strided convolution layers (sometimes called deconvolution layers). The output layer uses a tanh activation function. The autoencoder's weights are updated by a linear combination of two costs: the mean squared error between the original unmasked image and the reconstructed image, and binary cross entropy using the output of the discriminator (described in the Loss Functions section). A diagram of the autoencoder is shown below.

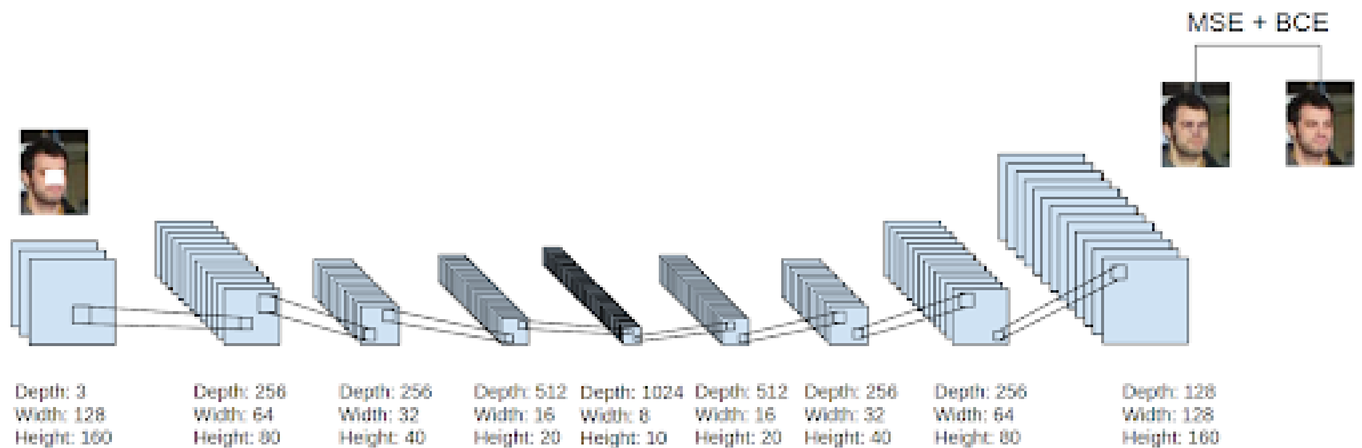


Figure 10

2.4.4 Discriminator

The discriminator consists of 4 convolutional layers. It accepts a 128x160 RGB image as input. The discriminator is trained to determine whether the input image is a real face. A sigmoid function is used on the final layer to yield a probability between 0 and 1. The probability represents the discriminator's belief that the input image is a real face. During training, the discriminator is updated using real images from the training set, and images generated from the autoencoder. The discriminator is rewarded for assigning a low probability to generated faces, and a high probability to real faces. A diagram of the discriminator is shown below.

2.4.5 DCGAN Guidelines

The architecture adheres to the following guidelines suggested in Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks which seemed to help make training stable:

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

2.4.6 Loss Function

There are four loss values defined in the model:

```
d_cost_real = binary_cross_entropy(p_real, T.ones(p_real.shape)).mean()
d_cost_gen = binary_cross_entropy(p_gen, T.zeros(p_gen.shape)).mean()

g_cost_d = binary_cross_entropy(p_gen, T.ones(p_gen.shape)).mean()
enc_cost = mse(source_flat, target_flat).mean()
```

The autoencoder's ability to fool the discriminator is determined by the discriminator's response when given a generated face. The autoencoder is also forced to learn the pixel wise representation of the faces by learning from the mean squared error from the ground truth images.

The discriminator's performance is measured by how well it can identify faces from ground truth images, and also how well it can recognize faces generated from the autoencoder.
The cost function used to update the autoencoder is:

```
g_cost_d + enc_cost * X
```

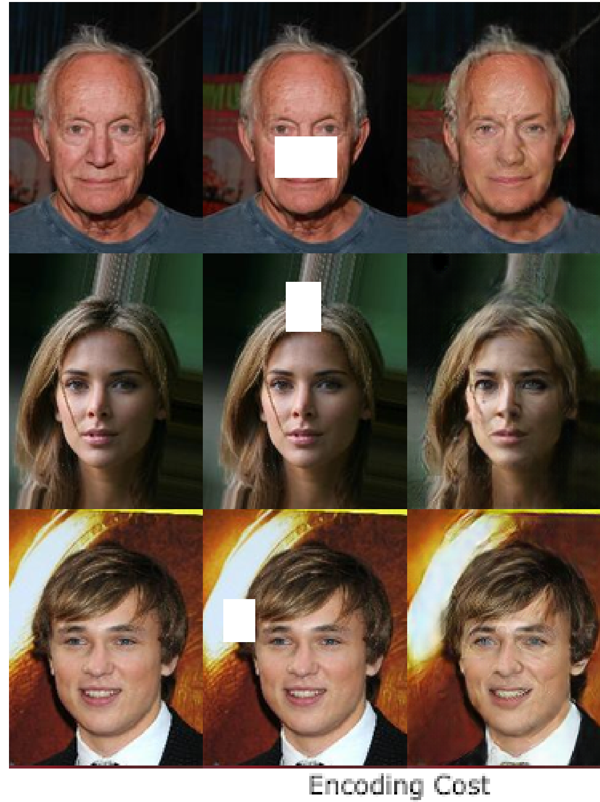
X is a parameter that decides how heavily to weigh MSE when calculating the autoencoder cost. The cost function used to update the discriminator is:

```
d_cost_real + d_cost_gen
```

During training and testing, they tracked: `g_cost_d`, `enc_cost` and `d_cost_real`

2.4.7 Result

Around 10k iterations, the encoding cost reaches a local minimum. A divergence trend starts at around 15k between the autoencoder's ability to fool the discriminator and the discriminator's ability of identifying real face data.



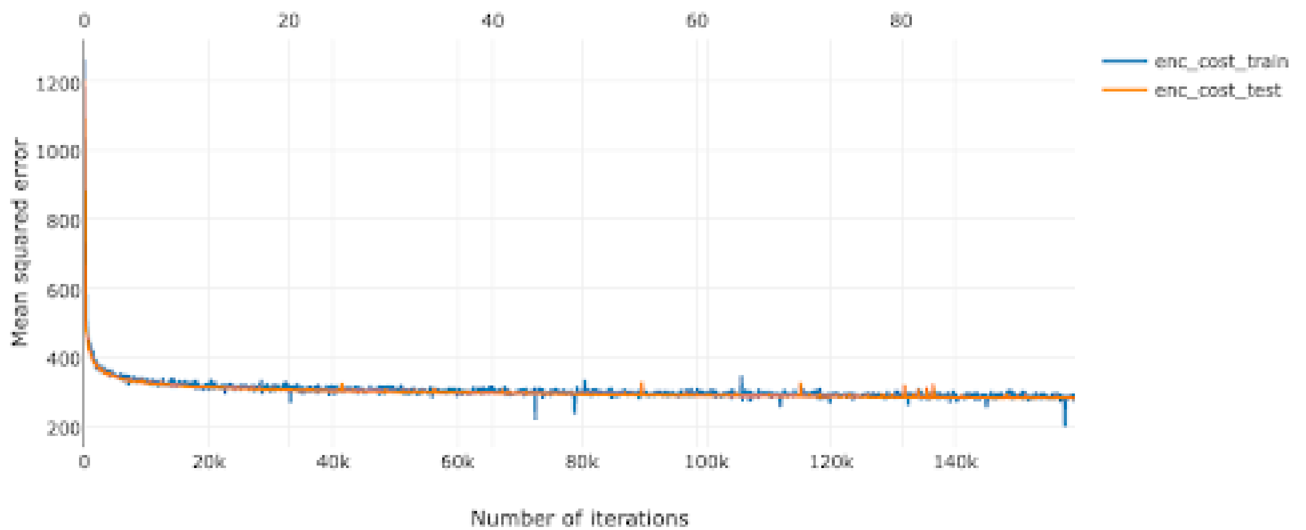


Figure 11

A possible explanation is that as the MSE encoding cost converges on a lower bound, the autoencoder is no longer able to fool the discriminator by updating with a loss function that so heavily favors the already converged MSE. A possible solution would be to use a dynamic learning mechanism to adjust the cost function for the autoencoder. As training progresses, the parameter X should be updated to weight MSE loss less once MSE cost has converged because learning from MSE no longer seems to be "efficient" in the sense of fooling the discriminator.

3 DBSCAN Clusterings

Clustering is the task of dividing the population or data points into a number of groups such that data points in the same groups are more similar to other data points in the same group than those in other groups. In simple words, the aim is to segregate groups with similar traits and assign them into clusters. As the figure shown below, we can use tools like k-means to find and separate different clusters:

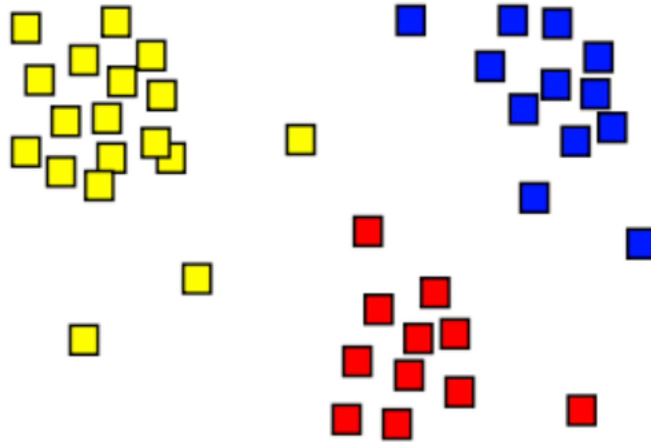


Figure 12: Clustering shown as the coloring of the squares into 3 clusters

Clustering groups a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters). It is a main task of exploratory data analysis, and a common technique for statistical data analysis, used in many fields, including pattern recognition, image analysis, information retrieval, bioinformatics, data compression, computer graphics and machine learning.

Clustering can be achieved by various algorithms that differ significantly in their understanding of what constitutes a cluster and how to efficiently find them. Popular notions of clusters include groups with small distances between cluster members, dense areas of the data space, intervals or particular statistical distributions. Clustering can therefore be formulated as a multi-objective optimization problem. The appropriate clustering algorithm and parameter settings (including parameters such as the distance function to use, a density threshold or the number of expected clusters) depend on the individual data set and intended use of the results. Cluster analysis as such is not an automatic task, but an iterative process of knowledge discovery or interactive multi-objective optimization that involves trial and failure. It is often necessary to modify data preprocessing and model parameters until the result achieves the desired properties.

There are basically three types of Clustering: Partition-based Clustering (like k-means, k-median), Hierarchical Clustering (like Agglomerative, Divisive), and Density-based Clustering (like DBSCAN).

3.1 DBSCAN Clustering

As the figure shown below, when dealing with a moon dataset like what we did in homework 2, it is hard for Partition-based Clustering like k-means to do clustering. The data points in the figure are grouped in arbitrary shapes or include outliers. Density-based clustering like DBSCAN will be much more efficient at finding high-density regions and outliers than k-means.

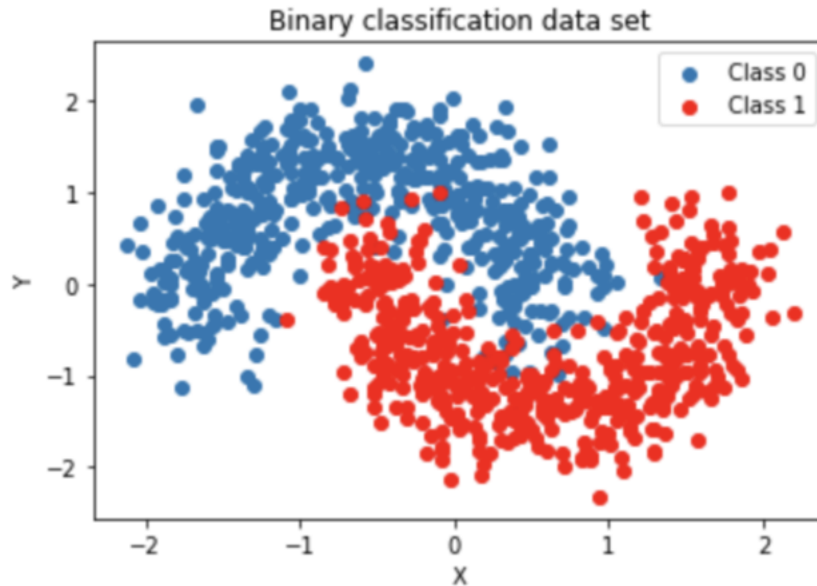


Figure 13: Moon Dataset from Keras

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is able to find arbitrary shaped clusters and clusters with noise (i.e. outliers). The idea is that a point belongs to a cluster if it is close to many points from that cluster. There are two key parameters of DBSCAN: eps (distance that specifies the neighborhoods) and minPts (minimum number of data points to define a cluster). There are three types of data points: Core point (a point is a core point if there are at least minPts number of points in its surrounding area with radius eps), Border point (a point is a border point if it is reachable from a core point and there are less than minPts number of points within its surrounding area), and Outlier (a point is an outlier if it is not a core point and not reachable from any core points).

As we can tell from the figure shown below, when the minPts is 4. Point A and the other red points are core points, because the area surrounding these points in an radius contain at least 4 points (including the point itself). Because they are all reachable from one another, they form a single cluster. Points B and C are not core points, but are reachable from A (via other core points) and thus belong to the cluster as well. Point N is a noise point that is neither a core point nor directly-reachable.

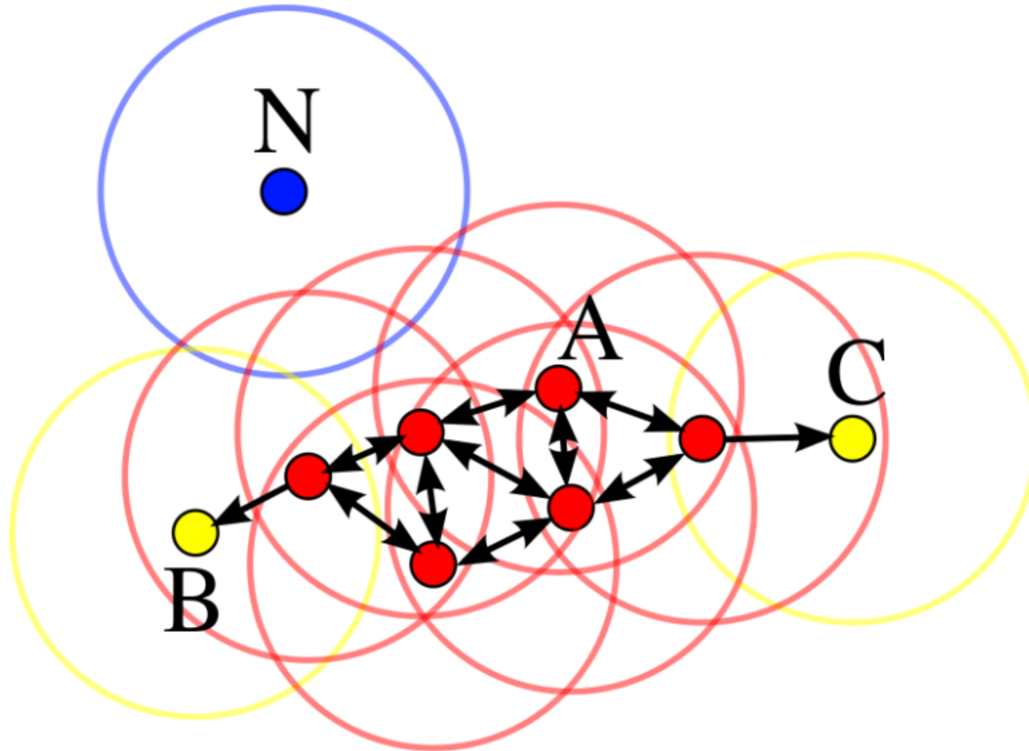


Figure 14: DBSCAN Clustering

Since minPts and eps are determined, a starting point is selected at random at it's neighborhood area using radius eps. If there are at least minPts number of points in the neighborhood, the point is marked as core point and a cluster formation starts. If not, the point is marked as noise. Once a cluster A formation starts, all the points within the neighborhood of the initial point become a part of cluster A. If these new points are also core points, the points that are in the neighborhood of them are also added to cluster A. After cluster A is filled, we randomly choose another point among the points that have not been visited in the previous steps and keep finding other points to fill the cluster. By applying these steps, DBSCAN is able to find high density regions and separate them from low density regions.

3.2 Evaluation of DBSCAN

DBSCAN visits each point of the database, possibly multiple times. For practical considerations, however, the time complexity is mostly governed by the number of regionQuery invocations. DBSCAN executes exactly one such query for each point, and if an indexing structure is used that executes a neighborhood query in O(logN), an overall average runtime complexity of O(N*logN) is obtained (if parameter is chosen in a meaningful way, i.e. such that on average only O(logN) points are returned). Without the use of an accelerating index structure, or on degenerated data (e.g. all points within a distance less than), the worst case run time complexity remains $O(N^2)$. The distance matrix of size $(N^2\text{-}N)/2$ can be materialized to avoid distance recomputations, but this needs $O(N^2)$ memory, whereas a non-matrix based implementation of DBSCAN only needs O(N) memory.

Since DBSCAN is a kind of Density-based Clustering, it not only does not require to specify the number of clusters beforehand, but also performs well with arbitrary shapes clusters, which makes it robust to outliers and able to detect the outliers.

However, in some application cases, determining an appropriate distance of neighborhood (eps) is not easy, which requiring another layer of information from the dataset. Besides, if clusters are very different in terms of in-cluster densities, DBSCAN is not well suited to define clusters. The characteristics of clusters are defined by the combination of eps-minPts parameters. Since we pass in one eps-minPts combination to the algorithm, it cannot generalize well to clusters with much different densities. Therefore, one of the original authors of DBSCAN published a refined version of hierarchical DBSCAN which no longer has the notion of border points. Instead, only the core points form the cluster, which can save some trouble of collecting the domain knowledge of the dataset.

References

[1] Clustering: Types of clustering: Clustering applications. Analytics Vidhya. (2020, October 18). Retrieved March 7, 2022, from https://www.analyticsvidhya.com/blog/2016/11/an-introduction-to-clustering-and-different-methods-of-clustering

[2] An introduction to autoencoders: Everything you need to know. An Introduction to Autoencoders: Everything You Need to Know. (n.d.). Retrieved March 8, 2022, from https://www.v7labs.com/blog/autoencoders-guidearchitecture

[3] Jeremy Jordan. (2018, July 16). Variational autoencoders. Jeremy Jordan. Retrieved March 8, 2022, from https://www.jeremyjordan.me/variational-autoencoders/

[4] Prakash, A. (2019, July 14). Different types of autoencoders. OpenGenus IQ: Computing Expertise amp; Legacy. Retrieved March 8, 2022, from https://iq.opengenus.org/types-of-autoencoder/

[5] Generative Adversarial Denoising Autoencoder for Face Completion. Avery Allen, Wenchen Li. https://faculty.cc.gatech.edu/ hays/7476/projects/Avery_Wenchen