MAX-PLANCK-INSTITUT
FÜR MOLEKULARE GENETIK

sc∗GMΔVÆ
sc∗GMVÆ

a Master Thesis in Bioinformatics

Thesis advisor / 1st Supervisor: professor Martin Vingron
2nd Supervisor: professor Tim Conrad

Yiftach Josef Kolb (Matrikelnummer 5195763)

Berlin, December 30, 2022

Freie Universität Berlin

Freie Universität Berlin

Declaration of Authorship

Disclaimer: Please note that only the German version of this document is legally binding. This translation is intended for the convenience of the non-German-reading public and is for informational purposes only.

Last name: Kolb	(Please use block capitals or typescript.)
First name: Yiftach	
Date of birth: 21.01.1981	
Student number: 5195763	

I hereby declare to Freie Universität Berlin that I have completed the following master thesis independently and without the use of sources and aids other than those cited.

I declare that the present work is free of plagiarism. Any statements that have been taken from other writings, whether directly or indirectly, have been clearly marked as such.

Further, I declare that this work has not been submitted to any other university as part of an examination attempt, either in identical or similar form, nor has it been published elsewhere.

Date: 19.12.2022 Signature: [illegible]

(______________________)

Abstract

Single cells RNASeq data (scRNASeq) can be thought of as coming from a conditional mixture distribution, where the categories indicate cell types and the conditions indicate either technical batches or post/pre–exposure effects. In this thesis we take the GMVAE (Gaussian mixture variational autoencoder) model [2], introduce some modifications to the underlying probabilistic model and test its suitability for analysis, embedding and generation of scRNASeq data. We think that the model is particularly suitable for (semi)supervised use case, where information on the cell types of the dataset is partially available. sc$*$GMVAE stands for "Single cell RNASeq ($*$) Conditional Gaussian Mixture Variation AutoEncoder".

Acknowledgement

I am very thankful to professor Martin Vingron for for being my mentor and advisor and for directing me through this work. I owe Martin, his department and the Max Planck institute a lot of gratitude for providing me a very welcoming, productive and scientifically rich environment. Thanks to professor Tim Conrad for being my thesis defense supervisor twice in a row!

List of Figures

Contents

Preface

This thesis could be colloquially summarized by: *VAEs – why should we (bioinormarticians) care about them.*

Chapter 1

Notations and definitions, preliminary concepts

1.1 Tensors, shape, axis, dimension

In machine learning one often encounters data structures that have multi-dimensional *shape* which we call *tensors*. For example a 28 over 28 color image can be represented as a 3-dimensional shape $(3, 28, 28)$ representing pixel's RGB channels (AKA its color), its height, and its width. This creates some confusion as to what one means by "dimensions"– the dimensions of the data shape which is 3, or the number of dimensions of the data content (for lack of a better term), which is $3 \times 28 \times 28$. For example a vector $\mathbf{x} \in \mathbb{R}^5$ is represented as a 1 dimensional shape but it has 5 *dimensions in total.* Similarly the color image has a 3 dimensional shape but it has $28 \cdot 28 \cdot 3$ dimensions in total. It has 3 *axes*, whose respective sizes are 28, 28, and 3.

A (real valued) *tensor* is an element of a tensor product space, for example the color image described above, $\mathbf{x} \in \mathbb{R}^{28\times 28\times 3} \triangleq \mathbb{R}^{28} \otimes \mathbb{R}^{28} \otimes \mathbb{R}^3$. A *tensor* has 0 or more axes and it generalizes scalar, vector, matrix and higher dimensional shaped entities.

In Pytorch [8] terminology *dimension* is used for the number of axes (data shape) but it is inconsistent with the way dimension is used in mathematics with regards to vectors which is the number of entries (data content size). We suggest that *dimension* should only refer to data content size, and *shape size* to the number of axes it comes in. So for the same $\mathbf{x}$ color image: $\dim(\mathbf{x}) = 3 \times 28 \times 28$ and $\text{shape}(\mathbf{x}) = (3, 28, 28)$ and $\text{shape size}(\mathbf{x}) = \text{axes}(\mathbf{x}) = 3$.

Definition 1.1. A *scalar* $x \in \mathbb{R}$ is an element of the (real) field. It has 0 *dimensions*, 0 *axes* and *shape* $(,)$.

A *vector* $\mathbf{x} \in \mathbb{R}^n$ has n dimensions, 1 axis, and shape $(n,)$.

A matrix $\mathbf{X} \in \mathbb{R}^{m\times n}$ has mn dimensions (in total), 2 axes, and shape of (m, n). Its first and second axes are said to be of sizes m and n respectively.

A tensor $\mathbf{x} \in \mathbb{R}^{n_1\times\cdots\times n_k}$ has $\prod_1^k n_i$ dimensions, k axes, and shape $(n_1, \dots, n_k)$. Its i's axis is said to be of size n_i.

In practice a tensor $\mathbf{x} \in \mathbb{R}^{n_1 \times \cdots \times n_k}$ is represented as a k dimensional array. We call its first axis the *row axis* or alternatively when we want to emphasize that this is a collection of several tensors, the *batch axis.* The tensor $\mathbf{x}_i = \mathbf{x}[i]$, which is the $k-1$ dimensional sub-array with the first coordinate held fixed, is called the *i'th row* of $\mathbf{x}$. Sometimes we want to represent a tensor $\mathbf{x}$ as a collections of tensors. For example it can represent a collection of several images. Each "row" then represents an image tensor.

It may be that our data set comes not as one tensor but in several tensors. For example we might have a tensor $\mathbf{x}$ representing an ordered set of images, and a tensor $\mathbf{y}$ which represents the category of each image. Because they have different shapes they don't fit together in one tensor. However since they refer to the same entities (images) their first axes have equal sizes. We use the notation $(\mathbf{x}, \mathbf{y})$ to denote the matching pairs of image/category, implicitly requiring them to have equally sized first axes.

If $\mathbf{x}, \mathbf{y}$ have the same number of axes and their respective axes sizes are equal on all but the last axis then we can concatenate them by "stacking" $\mathbf{y}$ on top of $\mathbf{x}$ along the last axis.

Definition 1.2. If $\mathbf{x}$ is a tensor, $\mathbf{x}_i$ represents the i'th "row" of $\mathbf{x}$, and $\mathbf{x} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ is the *row representation* or *batch representation* of $\mathbf{x}$.

Definition 1.3. Let $\mathbf{x}, \mathbf{y}$ be tensors whose first axes have equal dimensions, n. Then $(\mathbf{x}, \mathbf{y})$ is the set of ordered pairs $(\mathbf{x}, \mathbf{y}) \triangleq \{(\mathbf{x}_i, \mathbf{y}_i) : i = 1 \ldots n\}$.

Definition 1.4. If $\mathbf{x}, \mathbf{y}$ have the same number of axes and equal dimension on all but their last axis, the $(\mathbf{x}|\mathbf{y})$ is their concatenation on the last axis.

1.2 samples, batches, mean-sum rule

We distinguish between two types of tensors depending on what they represent. Let $\mathbf{x} \in \mathbb{R}^{m \times n \times l}$ be a tensor. If we say that $\mathbf{x}$ is a *sample* or a *data point* it means it is a single sample from our data set. If we say that it is a *batch*, then it represent a collection of m samples. In this case the first axis is the batch axis and the rest of the axes are the sample axes.

As a rule the default reduction is summation over sample axes and mean over the batch axes. For example if $\mathbf{x}$ is a sample, then $\|\mathbf{x}\|_1 = \sum_i \sum_j \sum_k |x_{i,j,k}|$ because it has no batch axis. If $\mathbf{x}$ is a batch, then we take the mean over the first axis: $\|\mathbf{x}\|_1 = \frac{1}{m} \sum_i \|\mathbf{x}_i\|_1 = \frac{1}{m} \sum_i \sum_j \sum_k |x_{i,j,k}|$.

The reason that we do that is that for batches, we want batches of different sizes to be comparable so it is straight forward to take mean. For the other axes, as we will see in the case of VAE we use the ELBO function where we have to sum over the sample axes.

1.3 Matrices and vectors

The type of data we work with in this paper can be represented as vectors. For example images of shape (h, w, c) can be flattened into a single axis shape $(h \cdot w \cdot c,)$ vector.

Throughout this paper (modulo typing errors) we use capital bold math Latin or Greek letters $(\boldsymbol{X}, \boldsymbol{\Sigma})$ to represent matrices. To stress that we talk about matrices rather than vectors we show product ($\times$) in the dimension, i.e $\boldsymbol{X} \in \mathbb{R}^{m\times n}$. Although technically the matrix–space is the tensor product $\mathbb{R}^m \otimes \mathbb{R}^n$.

Bold small math letters ($\boldsymbol{x}$) represent usually row vectors, but in cases where it makes sense may also represent matrices such as a batch of several vectors (each row is a different data point). In few occasions it makes sense to let it represent both a matrix and a vector, for example, $\boldsymbol{\sigma}$ may represent both the covariance matrix and the variance vector of a diagonal Gaussian distribution. Non-bold math letters (x, σ, ...) may represent scalar or vectors in some cases and hopefully it is clear from the context or explicitly stated.

Since we are only dealing with real matrices the transpose and the conjugation operators are the same ($A^T = A^*$) but over $\mathbb{C}$ conjugation is usually the "natural" operation and we use it to indicate that some property is still valid over $\mathbb{C}$ with conjugation.

Sometimes matrices are given in row/column/block notations inside brackets where the elements are concatenated in a way that makes positional sense. For example both $(\mathbf{x}, \mathbf{y})$ and $(\mathbf{x}|\mathbf{y})$ represent a matrix with 2 **columns**.

As mentioned usually just $\mathbf{x}$ means a column vector and $\mathbf{x}^T$ means a row vector but sometimes in matrix notation $\mathbf{x}$ represents a row when it makes sense. We use **curly** brackets to indicate the **row** representations of a matrix. For example $\{\mathbf{x}, \mathbf{y}\}$ represents a matrix whose **rows** are $\mathbf{x}$ and $\mathbf{y}$ (as row vectors), which alternatively could be represented as $(\mathbf{x}, \mathbf{y})^T$ (and symmetrically $(\mathbf{x}, \mathbf{y}) = \{\mathbf{x}, \mathbf{y}\}^T$).

$(\mathbf{X}, \mathbf{Y})$ or $(\mathbf{X}|\mathbf{Y})$ represent concatenation of two matrices along the second axis (concatenation of rows) which implicitly means they have the same number of rows. $\{\mathbf{X}, \mathbf{Y}\}$ represents concatenation along the first axis (concatenation of columns) which implies they must have equal number of columns.

Zero–blocks are indicated with 0 or are simply left as voids. For example $\begin{pmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \mathbf{0} & \boldsymbol{D} \end{pmatrix}$, $\begin{pmatrix} \boldsymbol{A} & \boldsymbol{B} \\ & \boldsymbol{D} \end{pmatrix}$ both represent block notation of the same upper–triangular matrix.

Definition 1.5. Let $\mathbf{X} = \{\mathbf{x}_1, \dots \mathbf{x}_m\} \in \mathbb{R}^{m\times n}$ be a matrix in **row** notation. Then its *squared Frobenius norm* is

$$\|X\|_F^2 \triangleq \operatorname{trace}(\mathbf{X}\mathbf{X}^*) = \sum_{i=1}^{m} \|\mathbf{x}_i\|_2^2 = \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}^2 \tag{1.1}$$

1.4 Functions and maps

Functions are usually understood to be scalar, namely $f : \mathbb{R}^n \to \mathbb{R}$ while maps are more general $g : \mathbb{R}^n \to \mathbb{R}^m$. When we say that a map (or function) $\phi : \mathbb{R}^n \to \mathbb{R}^m$ is *parameterized*, it implicitly means that ϕ has additional variables which we treat as parameters $\phi_{\mathbf{w}}(\mathbf{x}) = \phi(\mathbf{x}, \mathbf{w})$ where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{w}$ is the function's parameter set which we don't always specify its domain and we may not always subscript ϕ with it. The parameterized map ϕ itself may be identified with its parameter set and then both are designated with ϕ. It is important to stress out that the parameterization is not just an enumeration of the functions in our class, rather we want $\mathbf{w}$ to come from some domain and that the parameterized function $\phi(\mathbf{x}, \mathbf{w})$ should be *differentiable in its parameters* $\mathbf{w}$. It is of lesser concern whether ϕ is also differentiable in $\mathbf{x}$. For example a classifier is a discrete function over $\mathbf{x}$ so a neural network ϕ that classifies $\mathbf{x}$ is not differentiable in $\mathbf{x}$ but it is on its parameters $\mathbf{w}$.

In the context of neural networks, when we say *linear* map, we actually mean an *affine* map. An affine map $f(x_1 \dots x_n)$ can always be represented as a linear map with one extra variable which is held fixed $x_0 \equiv 1$: $f(x_0, \dots x_n) = b + a_1 x_1 + \dots a_n x_n$. We call b the *bias* of the linear map f.

1.5 Data types

We assume that the input data unless otherwise stated is real-valued matrix. Rows represent *samples* and columns represent *variables.* We assume that each row is a realization of a random vector. If we have N rows, then the corresponding N random vectors are assumed to be independent. So depending on the context, when we say observation, or row, we may mean the actual observed values, or to the random vector which was realized by said observation.

We deal with two kinds of datasets in this thesis. One of them is Single cell RNASeq data. This kind of data represents gene expression levels in individual cells, where rows represent cells and columns represent genes. So if we see a reading of 0.5 in row 2 column 4 in means that in cell 2 gene 4 has normalized expression of 0.5.

The other kind of data is images. For example the MNIST data set contains greyscale 28×28 images of hand written digits. We still think of such data set as a matrix. The first axis always represents the samples, so each "row" represent an image. The rest of the axes represent the image. Alternatively we can also flatten the images into one axis and think of an image as a row vector of $28 * 28$ dimensions.

There could possibly be additional data matrices with information about class or conditions. We use *one-hot encoding* to represent such information. For example in the case of the MNIST dataset every image also comes with a label which indicates what digit it is. Since there are 10 digits (0 to 9) the class matrix is going to have 10 columns and each row is a one-hot vector indicating the digit of the corresponding image.

Definition 1.6. A *data matrix* is a real–valued matrix $\boldsymbol{X} \in \mathbb{R}^{N \times n}$ which represents a set of N n-dimensional data points. The N rows are also called *observations* and the n columns are *variables.*

Definition 1.7. A *class matrix*, or also a *condition matrix* $\boldsymbol{C} \in \mathbb{R}^{\mathbb{N} \times c}$ is a real matrix which represents one-hot encoding of c classes or conditions over N samples. For example if sample i has class j, then $(\forall k \in 1, \dots, c)\boldsymbol{C}[i,k] = \delta_{jk}$.

We say that that $\boldsymbol{C}$ is a *class probability matrix* or a *relaxed class matrix* (same with condition) if instead of being one-hot its rows are distributions—each row is non-negative and sums up to 1.

Usually if the input data includes class/condition information, it comes as a class matrix (pure one-hot) but the output (the prediction) is naturally probabilistic and hence is relaxed.

1.5.1 Input set and target set

Our data usually comes in several, two or more matrices, and when it does, the rows of these matrices are paired (or tupled if they are more than 2). For example we could have an input data $\mathbf{X}$ and a target data $\mathbf{Y}$, representing samples from some unknown function $f(\mathbf{x}) = \mathbf{y}$ that we want to learn. This is generally how things are in an image classification task, where $\mathbf{X}$ may be a set of images, and $\mathbf{Y}$ is their labels. Implicitly $\mathbf{X}, \mathbf{Y}$ must have the same number of rows. And when we speak about paired input/target $\mathbf{x}, \mathbf{y}$ it means some $(\mathbf{x}, \mathbf{y}) \in (\mathbf{X}, \mathbf{Y})$ and $\mathbf{x}, \mathbf{y}$ belong to the same sample (same row number).

1.5.2 Probabilistic interpretation of the data

Suppose that we have a data matrix $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$. We think of $\mathbf{X}$ as a set of N independent samples, all drawn from the same data distribution $\mathbf{x} \sim p(\mathbf{x})$. We think of $\mathbf{x}_i$ as a realization of a random vector which we also denote with $\mathbf{x}_i$. The random vectors $\mathbf{x}_i$ are iid we designate with by $\mathbf{x}$ a 'generic' random vector with the same distribution. This is another motivation why we take mean over the batch dimension, because then $\|\mathbf{X}\| = \frac{1}{N}\sum_1^N \|\mathbf{x}_i\| \approx \mathbf{E}[\|\mathbf{x}\|]$.

1.6 Linear algebra preliminary: SVD and PCA

In the following we state some facts and bring without proof what are the singular value decomposition and the principle components of a matrix. For a full proof see [9].

Let $\boldsymbol{X} \in \mathbb{R}^{N \times n}$ be a real-valued matrix representing N samples of some n-dimensional data points and let $r = \operatorname{rank}(\boldsymbol{X}) \leq \min(n, N)$.

$\mathbf{X}\mathbf{X}^*$ and $\mathbf{X}^*\mathbf{X}$ are both symmetric and positive semi-definite (since we are only dealing with real-valued data $\mathbf{X}^* = \mathbf{X}^T$). Their eigenvalues are non-negative, and they both have the same positive eigenvalues, exactly r such, which we mark $s_1^2 \geq s_2^2 \geq \ldots s_r^2 > 0$. The values $s_1 \ldots s_r$ are called the *singular values* of $\boldsymbol{X}$.

$$\text{Let } \boldsymbol{S} = \begin{pmatrix} s_1 & & & \\ & s_2 & & \\ & & \ddots & \\ & & & s_r \end{pmatrix} \in \mathbb{R}^{r \times r}$$

Let $\boldsymbol{U} = (\boldsymbol{u}_1|\ldots|\boldsymbol{u}_N) \in \mathbb{R}^{N \times N}$ be the (column) right eigenvectors of $\mathbf{X}\mathbf{X}^*$ sorted by their eigenvalues. Then $\boldsymbol{U} = (\boldsymbol{U}_r, \boldsymbol{U}_k)$ where $\boldsymbol{U}_r = (\boldsymbol{u}_1|\ldots|\boldsymbol{u}_r) \in \mathbb{R}^{N \times r}$ are the first r eigenvectors corresponding to the non-zero eigenvalues, and $\boldsymbol{U}_k$ are the eigenvectors corresponding to the $N - r$ 0-eigenvalues. Similarly let $\boldsymbol{V} = (\boldsymbol{V}_r, \boldsymbol{V}_k) \in \mathbb{R}^{n \times n}$ be the (column) right eigenvectors of $\mathbf{X}^*\mathbf{X}$, sorted by the eigenvalues, where $\boldsymbol{V}_r = (\boldsymbol{v}_1|\ldots|\boldsymbol{v}_r) \in \mathbb{R}^{n \times r}$ are the firs r eigenvalues and $\boldsymbol{V}_k$ are the $n - r$ null-eigenvectors.

The critical observations is that $\boldsymbol{V}_r = \boldsymbol{X}^*\boldsymbol{U}_r S^{-1}$ and then $\boldsymbol{U}_r^*\mathbf{X}\boldsymbol{V}_r = S$.

The *singular value decomposition (SVD)* of $\mathbf{X}$ is

$$\mathbf{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^* \tag{1.2}$$

where $\boldsymbol{D} = \begin{pmatrix} \boldsymbol{S} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} \in \mathbb{R}^{N\times n}$ is diagonal.

$\boldsymbol{V}_r$ are called the *(right) principal components* of $\mathbf{X}$. Note that $\boldsymbol{V}_r^*\boldsymbol{V}_r = \boldsymbol{I}_r$ and that $\mathbf{X} = \mathbf{X}\boldsymbol{V}_r\boldsymbol{V}_r^* = (\mathbf{X}\boldsymbol{V}_r)\boldsymbol{V}_r^T$. If one looks at the second expression, it means that the each row of $\mathbf{X}$ is spanned by the orthogonal basis $\boldsymbol{V}_r^T$ (because the other vectors of $\boldsymbol{V}$ are in $\ker(\mathbf{X})$.

More generally For every $l \leq r$, let $\boldsymbol{V}_l \in \mathbb{R}^{N\times l}$ be the first l components, Then $\mathbf{X}\boldsymbol{V}_l\boldsymbol{V}_l^T$ is as close as we can get to $\mathbf{X}$ within an l-dimensional subspace of R^n, and $\boldsymbol{V_l}$ minimizes

$$\boldsymbol{V}_l = \text{argmin}_{\mathbf{W}}\{\|\mathbf{X} - \mathbf{X}\boldsymbol{W}\boldsymbol{W}^T\|_F^2 \quad : \quad \boldsymbol{W} \in \mathbb{R}^{n\times l}, \boldsymbol{W}^T\boldsymbol{W} = \boldsymbol{I}_l\} \tag{1.3}$$

Where $\|\cdot\|_F^2$ is simply the sum of squares of the matrix' entries.

If we consider the more general minimization problems:

$$\min_{\boldsymbol{E},\boldsymbol{D}}\{\|\mathbf{X} - \mathbf{X}\boldsymbol{E}\boldsymbol{D}\|_F^2 \quad : \quad \boldsymbol{E}, \boldsymbol{D}^{\boldsymbol{T}} \in \mathbb{R}^{n\times l},\} \tag{1.4}$$

$$\min_{\mathbf{W}}\{\|\mathbf{X} - \mathbf{X}\boldsymbol{W}\boldsymbol{W}^\dagger\|_F^2 \quad : \quad \boldsymbol{W} \in \mathbb{R}^{n\times l},\} \tag{1.5}$$

It can be shown [7] that the last two problems 1.4, 1.5 are equivalent and that for any solution E, D it must hold that $\boldsymbol{D} = \boldsymbol{E}^\dagger$. ($\boldsymbol{D}$ is the Moore–Penrose generalized inverse of $\boldsymbol{E}$). Moreover, $\boldsymbol{V}_l$ still minimizes the general problem 1.4 and for every solution $\boldsymbol{W}$, it must hold that $\text{span}\{\boldsymbol{W}\} = \text{span}\{\boldsymbol{V}_l\}$ (but it isn't necessarily an orthogonal matrix).

Chapter 2

Neural networks

We briefly discuss here some of the basics of neural network to provide clarity and motivation. Mostly based on [6].

2.1 Universal families of parameterized maps

If we take an expression such as $f_{a,b}(x) = ax + b$, if we hold (a, b) fixed on specific values, then we get a linear function on x. Every assignment of (a, b) defines a different linear function and in fact every linear function on one dimension can be uniquely described by these a and b. So we can say that $\{f_{a,b}\}_{a,b\in\mathbb{R}}$ is a *parameterization* of the class of all real linear functions on one variable. The distinction between what are the variables and what are the parameters is somewhat arbitrary and in the end, $f_{a,b}(x)$ is just another way to represent a 3-variable function $f(a, b, x)$. As we mentioned 1.4 sometimes we don't specify the parameters and we identify the parametrize map f with its parameter set as this set uniquely determines f.

As mentioned in the previous chapter, the way we parameterize a function is important. Given a parameterized map $\phi(\mathbf{x})$ we want ϕ to be differentiable in its parameters so that $\frac{\partial}{\partial \mathbf{w}}\phi$ exits for every parameter $\mathbf{w}$.

We call a class $\mathcal{F}$ of parameterized functions *universal* if every continuous function can be uniformly approximated (inside a bounded domain) by a function of that class. The class of all linear functions is not universal. But taking "any function" g is too general. What we actually want is a class of parameterized functions that is:

- as simple as possible to construct
- differentiable in both the parameters as well as the variables
- can uniformly approximate any continuous function in a bounded domain given sufficiently large set of parameters (i.e. is universal).

Still these requirements are not enough. For example, the class of multivariate polynomials can uniformly approximate any function. However it may not be a good idea to try to learn very complicated high dimensional data using polynomial representation. One reason is that the number of terms (monomials) grows very rapidly with the dimension and the degree of the polynomials: for n dimensions and m degrees there are something like $\binom{m+n}{m}$ monomial terms.

We want a universal class of simpler functions, that are almost as simple as linear, and yet that suits well for statistical learning. For example we want to be able to represent complicated functions with relatively few parameters. One such class of functions is the feed forward neural networks, which is the class of functions that are comprised from ”neurons”.

2.2 Neurons

Inspired from biology, a *neuron* is a many–to–one ($\mathbb{R}^n \to \mathbb{R}$) parameterized function which ”integrates” the input with a linear, or affine (see remark 1.4) function, and then applies a non-linear scalar function, which we call an *activation function*. In a sense it is the simplest function that is not linear. Moreover we only need one type of non-linear activation, e.g sigmoid, to construct arbitrarily complex neural networks. A degree 2 polynomial would be considered ”less simple” because it applies multiple non-linear multi-variable functions $x_i x_j \dots$.

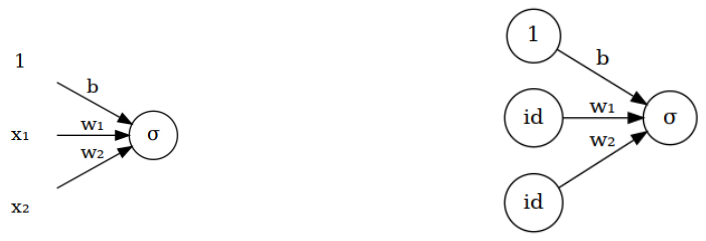
Definition 2.1. An *activation function* $\sigma : \mathbb{R} \to \mathbb{R}$ is any one of the following functions: $x \mapsto 1$ (constant), $x \mapsto x$ (identity) $x \mapsto \frac{e^x}{1+e^x}$ (sigmoid), and $x \mapsto \max(0, x)$ (ReLU).

σ can be applied on tensors by element-wise application. For example If $\mathbf{x} = (x_1, \dots x_n) \in \mathbb{R}^n$ then $\sigma(\mathbf{x})$ is the element-wise application $\sigma(\mathbf{x}) \triangleq (\sigma(x_1), \dots, \sigma(x_n))$.

In the official definition we narrowed it down to just 4 kinds but in general there are plenty of other activation functions. Also note that these functions have no parameters.

Definition 2.2. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be an activation function and let $f_{\mathbf{w}} : \mathbb{R}^n \to \mathbb{R}$ be a *parameterized* linear function. A *neuron* ν is the parameterized function $\nu = \nu_{\mathbf{w}} \triangleq \sigma \circ f_{\mathbf{w}}$.

The parameters $\mathbf{w}$ are called the *weights* of the neuron ν.



. Here the bias b is explicitly shown but usually it is not depicted. In the left the variable names are explicitly shown, while in the right they are not.

Figure 2.1: Two graphical descriptions of the neuron $\sigma(w_1x_1 + w_2x_2 + b)$

Connecting many neurons together can create powerful parameterized functions which we call neural networks. In feed forward networks the information only goes in one direction (no feedback) and as we will see it means the network is a directed acyclic graph.

Definition 2.3. A *feed forward neural network (NN)*is a **parameterized** map ϕ recursively defined follows:

1. Activation functions (1, *id*, and σ) are NNs which are called the *elementary neurons* and they have no parameters ($\mathbf{w} = \emptyset$).
2. neurons are NNs
3. If $\psi : \mathbb{R}^n \to \mathbb{R}^m$ is a parameterized linear map then it is a NN.
4. If $\psi : R^n \to \mathbb{R}^m$ and $\rho : \mathbb{R}^m \to \mathbb{R}^l$ are NNs and their parameter sets are disjoint then $\phi = \rho \circ \psi$ is a NN.
5. if $\nu : \mathbb{R}^n \to \mathbb{R}^m$ is a NN and if $\psi_i : \mathbb{R}^{k_i} \to \mathbb{R}^{n_i}, i = 1 \ldots l$ are NNs, such that $\sum_1^l n_i = n$ and if the parameter set of ν is disjoint from the combined parameters of the ψ_i's then $\phi = \nu(\psi_1, \ldots, \psi_n)$ is a NN .

The parameter set $\mathbf{w}$ is called the *weights* of ϕ. Often we don't distinguish between the network ϕ and its weights, and we identify both as ϕ.

In the definition we made the range and domain to be the entire $\mathbb{R}^n$ but it is not necessary, we just need for the composition to be valid.

Feed forward neural network are depicted as a directed acyclic graph where every node (with its incoming edges) corresponds to a neuron. You can think of figure 2.1 left as depicting the neuron "component" in a network, while figure 2.1 right shows a neural network description of single neuron, comprised from elementary neurons.

If rule 5 of the definition 2.3 is not used in the construction of ϕ, then the resulting network is hierarchical. Its graph can be partitioned into *levels* $l_0, l_1 \ldots$ and there are only directed edges between two consecutive levels $l_i \to l_{i+1}$ (see figure 2.2).

The label inside the neuron describes its activation function. In the diagrams, we let σ represent the sigmoid function. We represent the identity function either by the name of the variable (x_1, y etc.) it acts on or simple by *id*. We let the label 1 represent the constant function. We need the constant function because with it we can represent any affine map as a linear map with the first input always clamped to 1. But connecting 1 to every (non-input level) neuron would clutter the graph so it is not shown in most diagrams but still implicitly assumed. A directed edge between neurons means that the output of the neuron at the tail is multiplied by the edge weight and assigned to the input variable of the neuron it connects to. A Node's output is therefore only dependent on the output of its direct ancestral nodes (plus the bias which is usually not shown). Input-level neurons (sources) have no incoming edges and they represent the beginning of the computation. Output neurons (sinks) have no outgoing edges and their output is the final result of the computation.

It turns out [6] that the feed forward neural networks with a single type of non-linear activation (e.g. sigmoid) and a single hidden layer are "universal"; Which means that any continuous function f can be uniformly approximated by a feed forward neural network with a single hidden layer and Sigmoid as the non-linear activation function. More precisely, let $B \subseteq \mathbb{R}^n$ be a bounded domain. Let $f : B \to \mathbb{R}^m$ be continuous, and let $\epsilon \in (0, 1)$. Then there is a feed forward neural network with a single hidden layer $\phi = \phi_{\mathbf{w}}$ and there is some value assignment for the parameters $\mathbf{w}$ such that $(\forall \mathbf{x} \in B) \|\phi(\mathbf{x}) - f(\mathbf{x})\|_2 < \epsilon$. The size of that single hidden layer (the number of parameters) depends on f and ϵ.

In the definitions we only used linear maps to grow the network. There are other

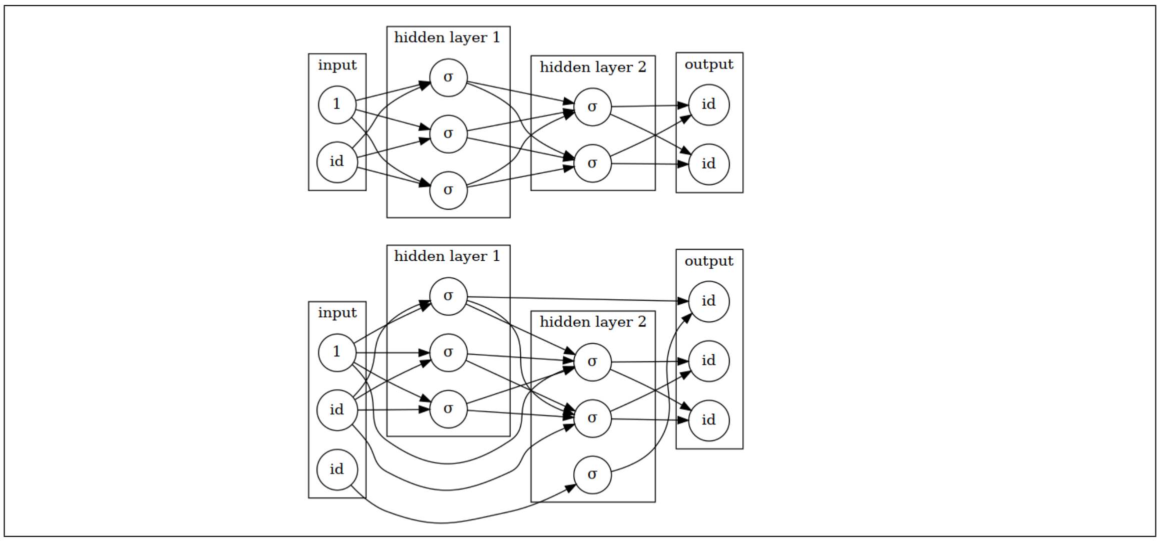


Figure 2.2: The network in the top didn't use rule 5 2.3 in the construction. It is strictly hierarchical and there are only edges between nodes of two consecutive layers. The one on the bottom is more general.

types of maps which are used, most commonly are convolutions but the principles and the graphical description remain essentially the same.

There are additional types of parameterized functions which are used ”within the layer” such as batch normalization but we won't get into that as this is not a thesis about neural networks per se.

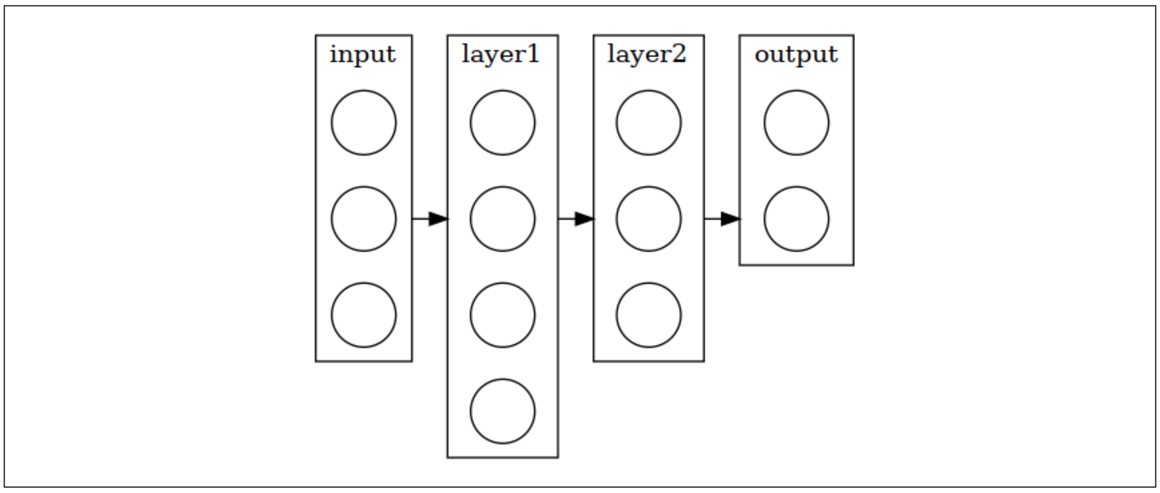


Figure 2.3: A graph of a hierarchical feed forward neural network where the connections are abstracted. Edges between layers may represent in this case a fully connected layer (every neuron has incoming edges from all neurons of the previous layer) but it could also be used for describing a convolution.

As figure 2.2 shows, The input layer is the where the input ($\mathbf{x}$) is ”fed in” and the output layer is the final result of the evaluation $\phi(\mathbf{x})$. We call all the layers (or neurons) that are not in the input level or the output level ”hidden” because we don't usually know what is the input/output value in these.

2.3 Loss functions

In the claim about neural networks being "universal" in terms of approximating function $f(\mathbf{x}) = \mathbf{y}$ with neural network $\phi(\mathbf{x})$. We stated specifically convergence in terms of l_2 norm $\|\phi(\mathbf{x}) - \mathbf{y}\|_2$, but the claim holds in theory and in practice with other types of "distance-like" functions which we call loss functions.

Moreover we usually don't know what is the function f which we try to approximate. Rather we are given paired samples of input/target $(\mathbf{x}, \mathbf{y})$ and we try to minimize the total error.

Definition 2.4. Let $\phi : \mathbb{R}^n \to \mathbb{R}^m$ be a neural network. A *loss function* is a differentiable function $\mathcal{L} : \mathbb{R}^m \oplus \mathbb{R}^m \to \mathbb{R}$. With "distance-like quality".

Typically the loss function is additive on the dimension, meaning it has the form $(\forall \mathbf{y}, \mathbf{z} \in \mathbb{R}^m)\mathcal{L}(\mathbf{y}, \mathbf{z})) = \sum_{i=1}^m \psi(y_i, z_i))$

Let $\mathbf{X} \in \mathbb{R}^{N\times n}, \mathbf{Y} \in \mathbb{R}^{N\times m}$ be the input and the target set and let $(\mathbf{x}, \mathbf{y})$ be a paired input/target. We use the loss function $\mathcal{L}$ as the target function for the minimization problem, $\min_{\mathbf{w}} \sum_{(\mathbf{x},\mathbf{y})} \mathcal{L}(\phi(\mathbf{x}), \mathbf{y})$ where the sum goes over all paires (N rows) (input, target).

For example $\mathcal{L}(\mathbf{y}, \mathbf{z})) = \|\mathbf{y} - \mathbf{z}\|_2^2 = \sum_i |y_i - z_i|^2$ is a one such loss function (the square error).

So far we defined ϕ and $\mathcal{L}$ on single input/target data points $\mathbf{x}$ and $\mathbf{y}$. But we are interested in minimizing the total error $\mathcal{L}(\phi(\mathbf{X}), \mathbf{Y})$. So first we need to state how these functions operate on sets of samples (matrices) rather than on data points (vectors).

Usually evaluation over the entire dataset is infeasible. Instead computation is performed on batches, which are relatively small chunks of the data.

Definition 2.5. Let $\boldsymbol{X} \in \mathbb{R}^{N\times n}$ be a data matrix. A *batch* $\boldsymbol{x} \in \mathbb{R}^{b\times n}$ is any subset of b rows of $\boldsymbol{X}$ (Note that in this case $\mathbf{x}$ represents a matrix).

Batch $\mathbf{x} = \{\mathbf{x}_1, \dots \mathbf{x}_b\} \in \mathbb{R}^{b\times n}$ (row notation) represents a subset of b samples out of the total of N samples in the dataset. Extending ϕ to operate on batches is trivial. $\phi(\mathbf{x}) = \{\phi(\mathbf{x}_i)\}$ is the matrix where ϕ is applied on the rows of the batch. Given an input batch $\mathbf{x}$ and corresponding target batch of $\mathbf{y}$, We extend the loss function to batches by averaging over the batch: $\mathcal{L}(\phi(\mathbf{x}), \mathbf{y}) \triangleq \frac{1}{b}\sum_{i=1}^b \mathcal{L}(\phi(\mathbf{x}_i), \mathbf{y}_i)$

Definition 2.6. Let ϕ be a neural network as defined in 2.3 and let $\mathcal{L}$ its associated loss function as defined in 2.4—over vectors. Let $\mathbf{x} = \{\mathbf{x}_1, \dots, \mathbf{x}_b\} \in \mathbb{R}^{b\times n}$ be a b-batch (in row notation) , and let $\mathbf{y} = \{\mathbf{y}_1, \dots, \mathbf{y}_b\} \in \mathbb{R}^{b\times m}$ be a corresponding target batch. Then ϕ and $\mathcal{L}$ *extended* over batches are:

$$\phi(\boldsymbol{x}) \triangleq \{\phi(\boldsymbol{x}_i)\}_{i=1}^b \in \mathbb{R}^{b\times m} \tag{2.1}$$

$$\mathcal{L}(\phi(\mathbf{x}), \mathbf{y}) \triangleq \frac{1}{b}\sum_{i=1}^b \mathcal{L}(\phi(\mathbf{x}_i), \mathbf{y}_i) \in \mathbb{R} \tag{2.2}$$

If $\mathcal{L}$ is the square error function $\|\cdot\|_2^2$ on vectors, then its extension to batches is $\frac{1}{b}\|\cdot\|_F^2$. The reason why we sum and don't average over the dimensions will be cleared later when we get into variational inference.

There is also a probabilistic way to interpret the total loss. We assume that the data points $\mathbf{X}, \mathbf{Y}$ were randomly and independently sampled from the unknown data distribution $p(\mathbf{x}, \mathbf{y})$. Then equation 2.2 can be reformulated as the expected loss [1]:

$$\mathcal{L}(\phi(\mathbf{X}), \mathbf{Y}) \approx \mathrm{E}_{\mathbf{x},\mathbf{y} \sim P(\mathbf{x},\mathbf{y})} \mathcal{L}(\phi(\mathbf{x}), \mathbf{y}) \tag{2.3}$$

2.4 Training

This is just a brief explanation of the basic principals. Training deep networks is a big subject which has many challenges and obstacles and a lot of heuristics are used.

Training the neural network ϕ_w means finding the weights that minimize the loss function applied on the training input/target paired sets $\mathbf{X}, \mathbf{Y}$, in other words minimizing $\min_w(\mathcal{L}(\phi_w(\mathbf{X}), \mathbf{Y})$. Usually we can't compute efficiently ϕ and $\mathcal{L}$ over the entire sets because N is too large, therefore we use batches.

Definition 2.7. Let ϕ_ω be a neural network and $\mathcal{L}$ its associated loss function. And let $(\mathbf{X}, \mathbf{Y})$ be our *training set* consisting of the data matrix $\mathbf{X}$ and $\mathbf{Y}$ the corresponding target matrix. Then *Training* of ϕ_ω with respect to $\mathcal{L}, \mathbf{X}$ means algorithmically approximating the minimization problem:

$$\min_\omega \mathcal{L}(\phi_\omega(\mathbf{X}), \mathbf{Y}) \tag{2.4}$$

During a *training step* the network is applied on a batch $(\mathbf{x}, \mathbf{y})$. Then the loss function is applied on the output of the network and a gradient (with relation to the weights) is taken using the efficient backpropagation algorithm [6]. The gradient is used for the weight update rule, which varies depending on the specific training algorithm. Typical training algorithms are SGD (stochastic gradient decent) and Adam [4], which is the one used throughout this work.

We only need to define the network, the loss function and the specific training algorithm. The rest (derivation, weight update etc.) is taken care for us by the backend of the software (Pytorch [8]) and can be regarded as a black box.

2.4.1 Training, validation and testing data sets

The data is partitioned into disjoint sets. The training set is used for the training of the model. The testing set is used for the final performance assessment. Sometimes a third subset, the validation set is used for tuning and tweaking the model during training. The point is that the model "doesn't know" the validation data because the weights are only trained on the training set, but the hyper-parameters are optimized based on the validation set. For example the validation set can be used for early stopping during the training. We didn't use a validation subset in our tests. Finally the assessment is performed on the testing set which was completely held out during the training and hyper-parameter tuning.

2.4.2 Un/Supervised learning

In unsupervised learning one seeks to ”learn” or infer the target set **Y** (for example category information) from **X** without seeing **Y** during training. For example in the case of MNIST we want to teach the model to distinguish 10 categories of images corresponding to the 10 digits, without having access to the digit tags in the training set.

Supervised learning means the **Y** target information is fully accessible (every image is tagged with the digit it represents). This is a much simpler classification task.

Semisupervised learning is the hybrid case of both, where the training set includes a small portion of known paired input/targets $(\mathbf{x}, \mathbf{y})$ while for the rest of the training set we only have $\mathbf{x}$ input and need to infer $\mathbf{y}$. Semisupervised learning tasks often arise in natural situations. For example there may be a large image data set where only a portion of the images have been manually tagged.

2.5 Autoencoders

The basic type of an autoencoder which we informally call ”vanilla” autoencoder is a neural network that tries to ”learn” the identity function. Though it sounds pointless on a first thought, the point is how we construct this network. An autoencoder consists of two neural networks. An encoder network maps the input into a lower dimensional so called ”latent space”, and a decoder network maps the latent space back into the high dimensional input layer. In the case of the vanilla autoencoder the target for the loss function is the same as the input $\mathbf{Y} = \mathbf{X}$.

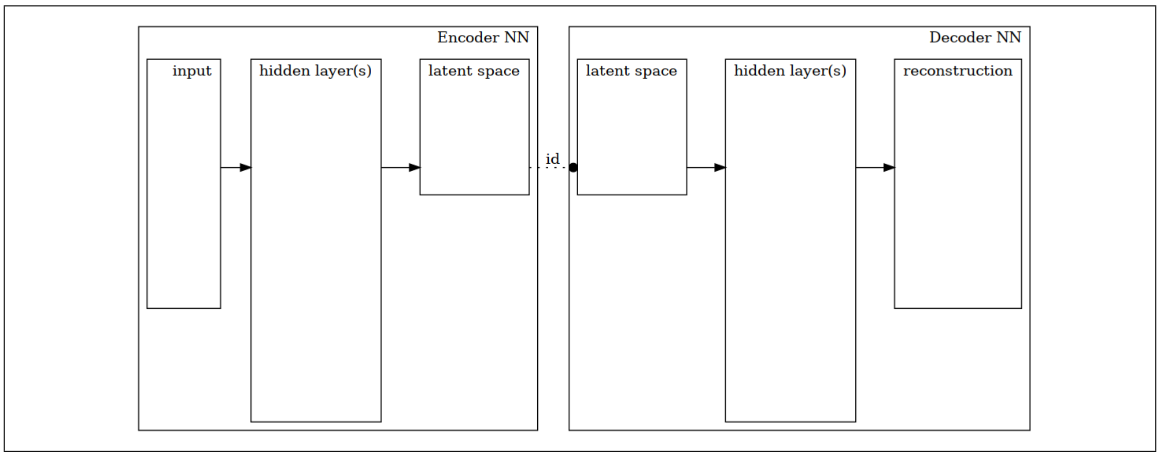


Figure 2.4: A graphic description of a ”vanilla” autoencoder.

Definition 2.8. An *Autoencoder* (AE) is a pair (ϕ, ψ) of feed forward neural networks $\psi : \mathbb{R}^n \to \mathbb{R}^m, \nu : \mathbb{R}^m \to \mathbb{R}^n$.

ψ is called the *encoder* network, and ν is called the *decoder* network and the composition $\phi = \nu \circ \psi$ is called the *autoencoding network.*

We call $\mathbb{R}^m$ or more genrally the domain of the decoder, *the latent space*, and $\mathbb{R}^n$ (or more generally the domain of the encoder) is called *the observed space.*

Given a batch $\mathbf{x} \in \mathbb{R}^{b \times n}$ we call $\mathbf{z} = \psi(\mathbf{x}) \in \mathbb{R}^{b \times m}$ the *latent representation* of $\mathbf{x}$ or the *encoding* of $\mathbf{x}$.

While the definition as is given is symmetric, it is assumed that $n > m$, and therefore ψ represents dimensional reduction (in other words encoding) of the data and ν represents expansion back to original space (decoding).

The idea here is that the original high dimensional data can be embedded in a low dimensional space by the encoder. The decoder then can reconstruct the original data from the embedding.

There are many variations of autoencoders. For example a "denoising" autoencoder is essentially that same model but it receives a "noisy" version of the input and tries to reconstruct the original clean version. We informally call the type of autoencoder of definition 2.8 which aims to learn the identity function on the original input, and which is using the square error loss function, a "vanilla" autoencoder.

2.5.1 A word choosing latent space dimension and plotting it

The dimension of the latent space is a hyperparameter of the model. It's choice can affect the model's performance and its numerical stability. We don't claim that we some sort of a system of how to choose it. For us it's a matter of trial and error with some common sense. For example it should be in the same order as PCA dimensions (which defaults on 50). We usually choose something in the range 8—64.

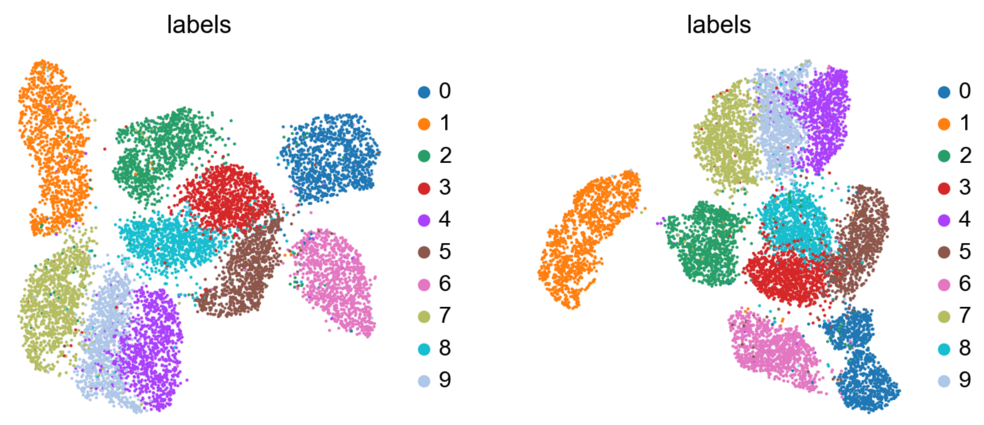
When it comes to plotting the latent space, one option is to set the dimension on 2 or 3 so it can be plotted directly. This works ok for some datasets but generally we need more freedom in choosing this value. And in the higher dimension case, first we perform PCA on the latent space. Then either directly plot the 2 or 3 most significant PCs, or use UMAP [5]. UMAP seems to be the golden standard nowadays, and it is used both in the machine learning as well as in the RNASeq milieus. For us UMAP is a black box that projects input into 2 dimensions relatively faithfully and does it fast (unlike TSNE). In some other cases, we show UMAP of the input space $\mathbf{X}$ rather than of latent space. In each image it should be stated which space and which projection is used.

2.5.2 Relation between PCA and AE

For **centered** data, where every variable (column of $\boldsymbol{X}$) has a sample mean of 0, the first $k \leq \text{rank}(\boldsymbol{X})$ principle components $\boldsymbol{P}$ are the solution for equation 1.3; Whereas a **linear** autoencoder solves equation 1.5. As mentioned, it must hold that $E = D^{\dagger}$ (the encoder must be the Moore-Penrose inverse of the decoder).

A linear autoencoder with the square error loss function is almost equivalent to PCA [7]; At the optimum, a bottleneck space of dimension k is spanned by the first k principle components of the input $\boldsymbol{X}$. In general, an AE can be seen a PCA-like, but non-linear method for dimensionality reduction.

Figure 2.5 shows on the left a UMAP [5] of the principle components of the testing subset of MNIST (images of hand written digits). On the left we see a UMAP of the latent space encoded by the encoder of a "vanilla" autoencoder with the square error loss function. The autoencoder was trained on the training set and didn't "see" the testing images during training. The results appear quite similar.



(a) MNIST: UMAP of the PCA (b) MNIST: UMAP of the AE

Figure 2.5: PCA (l) compared with ”vanilla” autoencoder (r) on the MNIST dataset

These figures figs. 2.1 and 2.5

See figure 2.1

This citations [1, 3] is based.

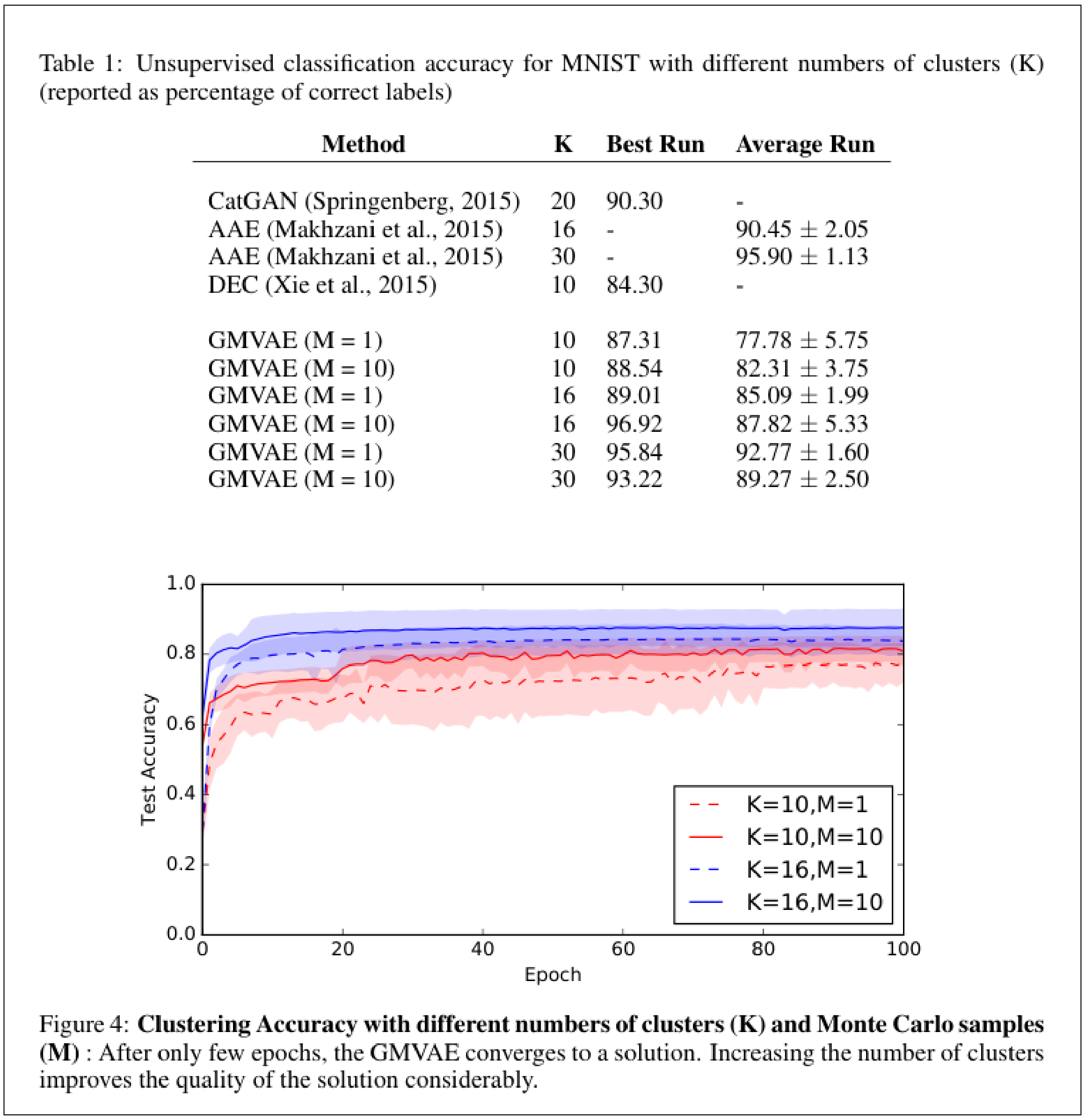


Table 1: Unsupervised classification accuracy for MNIST with different numbers of clusters (K) (reported as percentage of correct labels)

Method	**K**	**Best Run**	**Average Run**
CatGAN (Springenberg, 2015)	20	90.30	-
AAE (Makhzani et al., 2015)	16	-	90.45 ± 2.05
AAE (Makhzani et al., 2015)	30	-	95.90 ± 1.13
DEC (Xie et al., 2015)	10	84.30	-
GMVAE (M = 1)	10	87.31	77.78 ± 5.75
GMVAE (M = 10)	10	88.54	82.31 ± 3.75
GMVAE (M = 1)	16	89.01	85.09 ± 1.99
GMVAE (M = 10)	16	96.92	87.82 ± 5.33
GMVAE (M = 1)	30	95.84	92.77 ± 1.60
GMVAE (M = 10)	30	93.22	89.27 ± 2.50

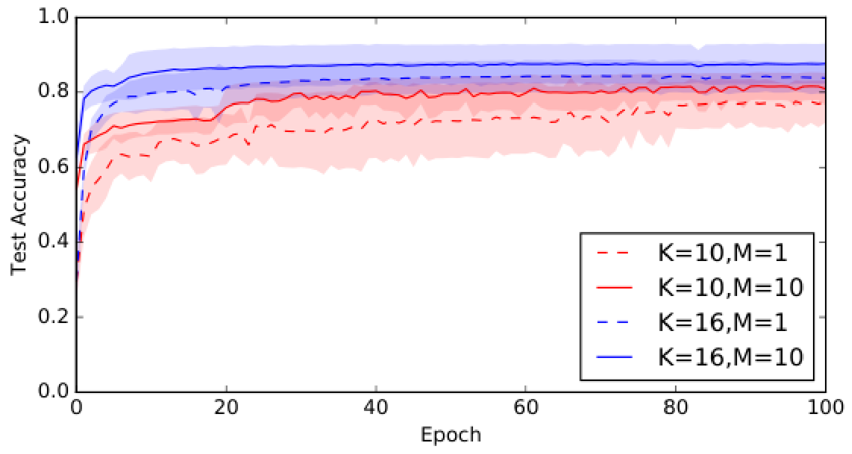


Figure 4: **Clustering Accuracy with different numbers of clusters (K) and Monte Carlo samples (M)** : After only few epochs, the GMVAE converges to a solution. Increasing the number of clusters improves the quality of the solution considerably.

Figure 2.6: Comparison of results by GMVAE and other methods as reported by [2]

Bibliography

[1] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning.* Vol. 4. 4. Springer, 2006.

[2] Nat Dilokthanakul et al. "Deep unsupervised clustering with gaussian mixture variational autoencoders". In: *arXiv preprint arXiv:1611.02648* (2016).

[3] Xifeng Guo et al. "Improved deep embedded clustering with local structure preservation." In: *Ijcai.* 2017, pp. 1753–1759.

[4] Imran Khan Mohd Jais, Amelia Ritahani Ismail, and Syed Qamrun Nisa. "Adam optimization algorithm for wide and deep neural network". In: *Knowledge Engineering and Data Science* 2.1 (2019), pp. 41–46.

[5] Leland McInnes, John Healy, and James Melville. "Umap: Uniform manifold approximation and projection for dimension reduction". In: *arXiv preprint arXiv:1802.03426* (2018).

[6] Michael A Nielsen. *Neural networks and deep learning.* Vol. 25. Determination press San Francisco, CA, USA, 2015.

[7] Elad Plaut. "From principal subspaces to principal components with linear autoencoders". In: *arXiv preprint arXiv:1804.10253* (2018).

[8] Automatic Differentiation In Pytorch. *Pytorch.* 2018.

[9] Denis Serre. "Matrices: Theory & Applications Additional exercises". In: *L'Ecole Normale Supérieure de Lyon* (2001).