c∗GMΔVÆs—q

a Master Thesis in Bioinformatics

Advisor / supervisor: Professor Martin Vingron
Supervisor: Professor Tim Conrad

Yiftach Josef Kolb (Matrikelnummer 5195763)

Berlin, January 4, 2023

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: _____

(_____)

Abstract

Single–cell–RNA–Sequencing 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. This thesis takes the GM-VAE (Gaussian mixture variational autoencoder) model [2], and introduces some modifications to its underlying probabilistic model. The resulting model is named (using regex) $c*GM\Delta V\text{Æ}$. The model's applicability in various scenarios for analysis, embedding, and generation of scRNASeq data is tested. It is particularly suitable for the (semi)supervised use case, where individual cell type information is partially available in the dataset or exists already in another, similar dataset. The model's name and the thesis' title c∗GMΔ *VÆs—q* stand for: (maybe Conditional) Gaussian Mixture with Dirichlet prior Variational Auto Encoder (for) s(ingle–cell-RNA–Se)q.

Acknowledgement

I am very thankful to Professor Martin Vingron for being my mentor and advisor, and for directing me through this work. A big thank you and much gratitude to Martin, his department, and the Max–Planck–Institute for molecular genetics, for providing the very welcoming, productive and scientifically rich environment. Many thanks to Professor Tim Conrad for being my thesis reviewer.

List of Figures

Contents

Preface

VAEs – why should we (bioinormarticians) care about them?.

When professor Vingron and I originally set on a theme for exploration towards a thesis, it was roughly about researching autoencoders and how they can be applied to single cell RNASeq (scRNASeq) data. I knew a little bit about autoencoders and neural networks and little to nothing about scRNASeq. It soon became clear that we are dealing with datasets of gene expression levels for individual cells. Each cell is supposed to have an expression profile depending on mostly cell type and also other factors such as cell cycle and , but cell type is the first order of approximation.

While doing preliminary research into neural networks I was thinking, since cell types is categorical data, I am dealing with a classification task, unsupervised learning, something like that; How can an autoencoder help me with that. So I've non-exhaustively searched what has been done with autoencoders on scRNASeq and similar data; Basically it's about replacing PCA with an AE, VAE or some fancier derivative. The rest of the analysis is with unrelated methods. Another use is for batch effect reduction by way of a conditional model. But VAEs can do more than that.

The entire construct that is VAE was invented, as far as I know, by Kingma and Welling [9, 10]. About one follow–up from their original paper they introduce semi–supervised learning. Nice. But their models doesn't work well on unsupervised learning. Then a little bit more research got me into the Gaussian mixture model VAE of Dilokthanakul [2], where they demonstrate unsupervised learning on some test datasets like MNIST. I settled on it as the base model to experiment with. By the end of this thesis, my goal is to provide a skeptical overview of what these models are good for as well as their limitations.

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 are called *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 [20] 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). The author proposes that *dimension* should only refer to data content size, and *shape size* should refer to the number of axes it comes in. 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 $(n_1, \ldots, n_k)$. 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. Its first axis is called 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 datasets used 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) capital bold math Latin or Greek letters $(\boldsymbol{X}, \boldsymbol{\Sigma})$ represent matrices. To stress that we talk about matrices rather than vectors,

the dimension is displayed as product ($\times$), 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. **Curly** brackets 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 there could be 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 $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$. 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 [23].

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 $r = \operatorname{rank}(\boldsymbol{X}) \le \min(n, N)$.

$\mathbf{X}\mathbf{X}^*$ 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 $s_1^2 \ge s_2^2 \ge \dots s_r^2 > 0$. The values $s_1 \dots s_r$ are called the *singular values* of $\boldsymbol{X}$.

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

Let $\boldsymbol{U} = (\boldsymbol{u}_1|\dots|\boldsymbol{u}_N) \in \mathbb{R}^{N\times N}$ be the (column) right eigenvectors of $\mathbf{X}\mathbf{X}^*$ sorted by their eigenvalues. Then $\boldsymbol{U} = (\boldsymbol{U}_r, \boldsymbol{U}_k)$ where $\boldsymbol{U}_r = (\boldsymbol{u}_1|\dots|\boldsymbol{u}_r) \in \mathbb{R}^{N\times r}$ 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 $\boldsymbol{V}_r = (\boldsymbol{v}_1|\dots|\boldsymbol{v}_r) \in \mathbb{R}^{n\times r}$ 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:

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

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

It can be shown [19] 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 [18].

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 $\sigma(w_1 x_1 + w_2 x_2 + b)$

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 $\psi_i : \mathbb{R}^{k_i} \to \mathbb{R}^{n_i}, i = 1 \dots l$ 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 $\phi = \nu(\psi_1, \dots, \psi_n)$ 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* $l_0, l_1 \dots$ 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 [18] 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

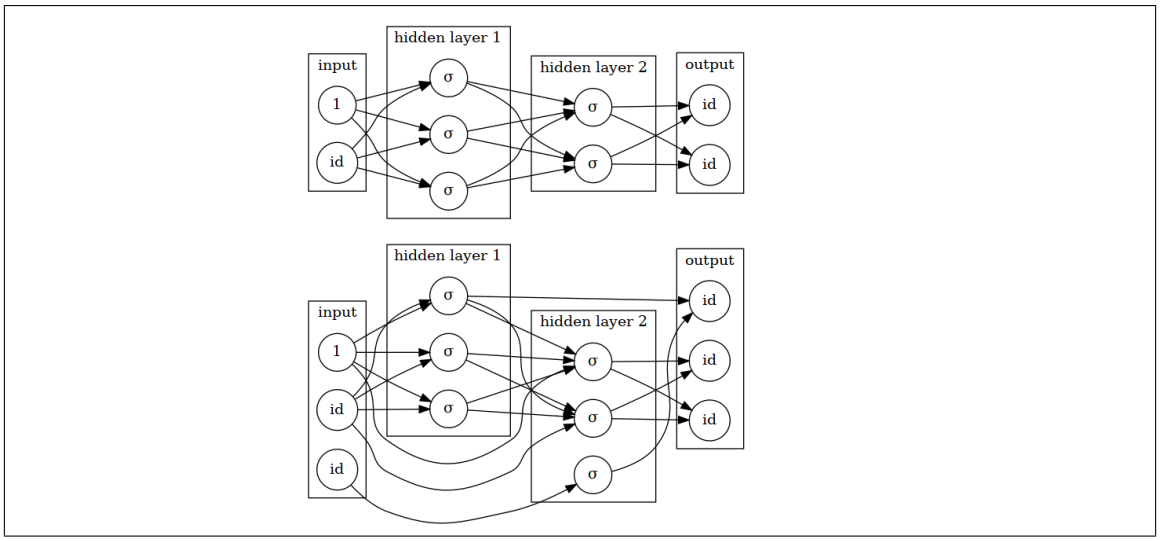


Figure 2.2: The network in the top didn't use rule 5 2.3 in the construction. It is strictly hierarchical and there are only edges between nodes of two consecutive layers. The one on the bottom is more general.

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.

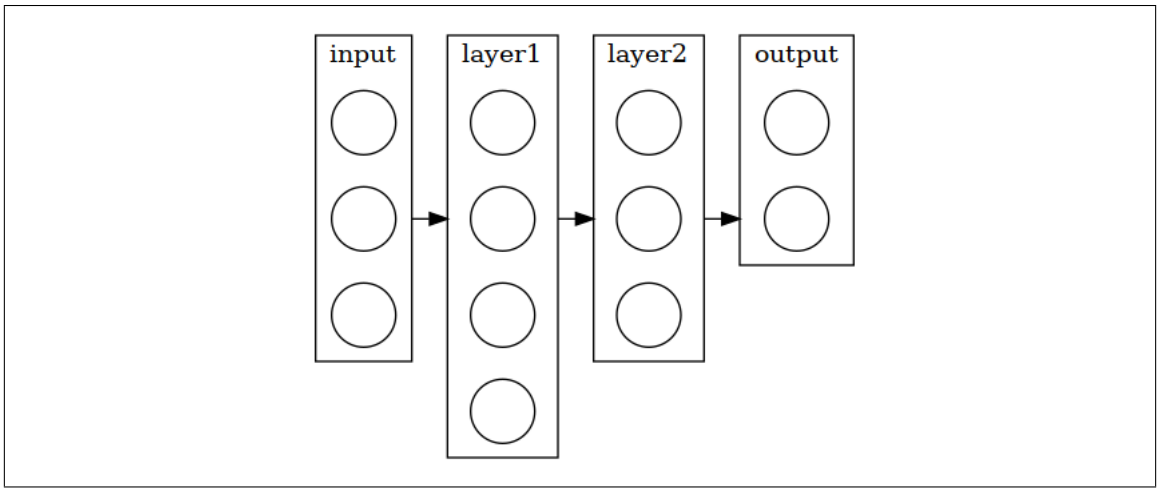


Figure 2.3: A graph of a hierarchical feed forward neural network where the connections are abstracted. Edges between layers may represent in this case a fully connected layer (every neuron has incoming edges from all neurons of the previous layer) but it could also be used for describing a convolution.

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 [18]. 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 [6], 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 [20]) 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 $\mathbf{Y}$ (for example category information) from $\mathbf{X}$ without seeing $\mathbf{Y}$ 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 $\mathbf{Y}$ 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 $\mathbf{Y} = \mathbf{X}$.

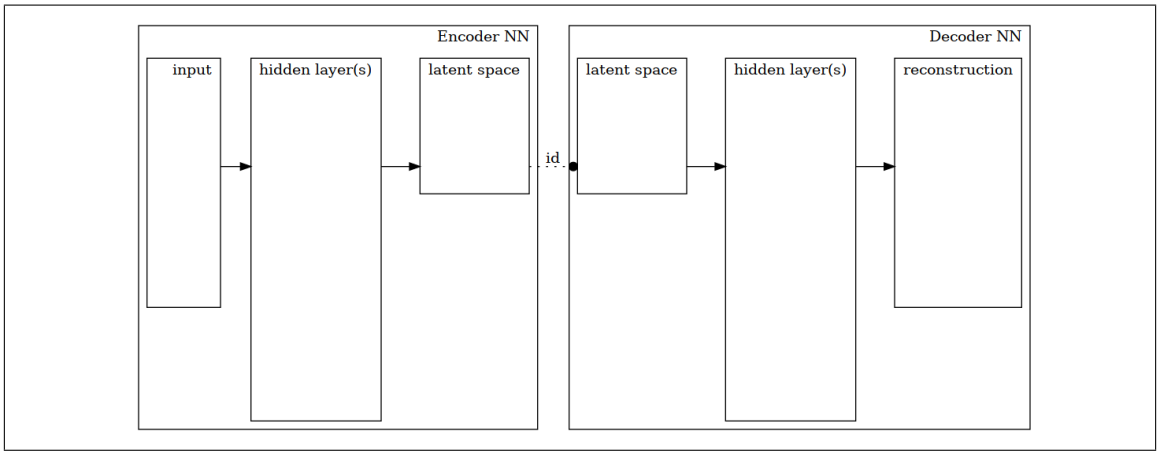


Figure 2.4: A graphic description of a "vanilla" autoencoder.

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.

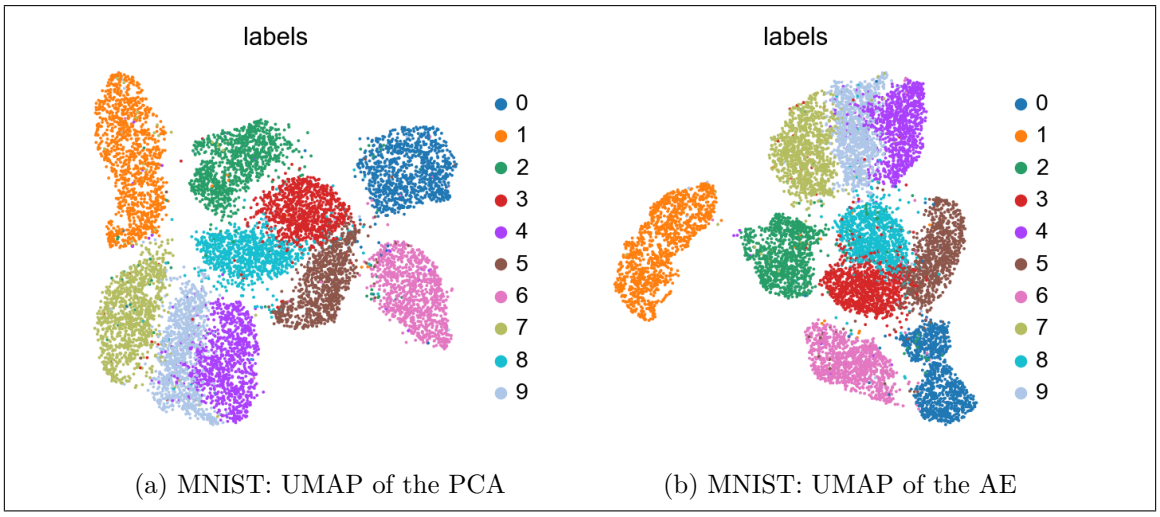
When it comes to plotting the latent space, one option is to set the dimension on 2 or 3 so it can be plotted directly. This works ok for some datasets but generally we need more freedom in choosing this value. And in the higher dimension case, first we perform PCA on the latent space. Then either directly plot the 2 or 3 most significant PCs, or use UMAP [17]. 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 [19]; 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 [17] 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

Chapter 3

Variational inference and variational autoencoders

3.1 Variational Inference

Here we briefly explain the idea behind variational inference and introduce the ELBO which is the loss function we'll use throughout this text. For more details see [1].

We treat the data matrix as a set of independent observations (its rows) $\boldsymbol{X} = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_N\}$ which we try to explain by a probabilistic model. Each row $\mathbf{x}_i$ is considered a realization of a random vector, which we also denote by $\mathbf{x}_i$ (as explained in the notation section) and similarly $\mathbf{X}$ represents both the set of random vectors as well as the realization.

We assume that the $\boldsymbol{x}_i$'s are independent and identically distributes (i.i.d) random vectors with some distribution function $\mathbf{x} \sim p(\mathbf{x})$ and therefore for the entire dataset it holds that $p(\boldsymbol{X}) = \prod p(\boldsymbol{x}_i)$.

Definition 3.1. Let $\boldsymbol{X} \in \mathbb{R}^{N \times n}$ be a data matrix and let $\{\boldsymbol{x}_i\}_1^n$ be its rows, which we assume to be i.i.d with some (unknown) distribution $\mathbf{x} \sim p(\boldsymbol{x})$. Then $\log p(\boldsymbol{X}) = \sum_1^N \log p(\boldsymbol{x}_i)$ is called the *log evidence* of our data.

$\frac{1}{N} \log p(\mathbf{X})$ is the *mean log evidence* (remember the mean-sum rule for data sets and batches 1.2).

An observation random vector $\boldsymbol{x}$ is high dimensional however we have some reason to believe that behind the scenes there is some hidden (latent), smaller dimensional random vector $\mathbf{z}$ that generates it. In other words we think that $\mathbf{x}$ is conditioned on $\mathbf{z}$ and we can speak of the joint distribution:

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z})$$

. And again $\mathbf{x}_i$ only depends on $\mathbf{z}_i$ so everything factors nicely, e.g. $p(\mathbf{X}|\mathbf{Z}) = \prod_i p(\mathbf{x}_i|\mathbf{z}_i)$. Informally, $\mathbf{z}$ is a simpler variable which "explains" $\mathbf{x}$ and can be used to for example, to classify $\mathbf{x}$. For this reason we are interested in knowing the flipped conditional distribution $p(\mathbf{z}|\mathbf{x})$.

Suppose that we have a fully Bayesian model. In this case there are no parameters because the parameters are themselves stochastic variables with some suitable priors.

We can therefore pack all the latent variables and stochastic parameters into one latent "meta variable" $\mathbf{z}$, which is some multidimensional random vector and which is possibly composed of several simpler random vectors (for example a categorical and a normal random vectors). We similarly pack all the observed variables into one meta variable $\boldsymbol{x}$. Together we have a distribution $p(\boldsymbol{x}, \boldsymbol{z})$ and the working assumption is that it is easy to factorize $p(\boldsymbol{x}, \boldsymbol{z}) = p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})$, however $p(\boldsymbol{z}|\boldsymbol{x})$ is intractable and $p(\boldsymbol{x})$ is unknown.

We are being Bayesian here so we consider $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots\}$ to be a constant set of observations and we want to best explain $p(\boldsymbol{X})$ by finding as high as possible lower bound for it (or rather to $\log p(\boldsymbol{X})$, the *log evidence*). A second goal is to approximate the intractable $p(\boldsymbol{z}|\boldsymbol{x})$ by some simpler distribution $q(\boldsymbol{z})$ taken from some family of distributions.

Definition 3.2. Let $\boldsymbol{x}, \boldsymbol{z}$ be random variables with joint distribution $p(\boldsymbol{x}, \boldsymbol{z})$ and let $q(\boldsymbol{z})$ be any distribution. Let $(\mathbf{X}, \mathbf{Z}) = \{(\mathbf{x}_1, \mathbf{z}_1), \dots, (\mathbf{x}_N, \mathbf{z}_N)\}$ be N independent replications of $(\mathbf{x}, \mathbf{z})$. The *evidence lower bound (ELBO)* with respect to p, q is:

$$-\mathcal{L}(q, p, \mathbf{x}) \triangleq \int \log \frac{p(\boldsymbol{x}, \boldsymbol{z})}{q(\boldsymbol{z})} dq(\boldsymbol{z}) \tag{3.1}$$

$$-\mathcal{L}(q, p) \triangleq -\mathcal{L}(q, p, \mathbf{X}) = \frac{1}{N} \sum_{1}^{N} (-\mathcal{L}(q, p, \mathbf{x}_i) \tag{3.2}$$

$$\approx \mathbf{E}_{\mathbf{x}}[-\mathcal{L}(q, p, \mathbf{x})] \tag{3.3}$$

Equation 3.2 is no longer treated as a function of $\mathbf{X}$ because it is taken over all of our data which we think of as a constant. The reason that we mark the ELBO with $-\mathcal{L}$ is because we use the minus ELBO, $\mathcal{L}$, as the loss function for VAEs.

The following equation shows that the *ELBO* is a lower bound for the *mean log evidence*. (using Jensen inequality)

$$\begin{aligned}
\frac{1}{N} \log p(\boldsymbol{X}) &= \frac{1}{N} \log \int p(\boldsymbol{X}, \boldsymbol{Z}) d\boldsymbol{Z} && \text{taking marginal} \\
&= \frac{1}{N} \log \int \frac{p(\boldsymbol{X}, \boldsymbol{Z})}{q(\boldsymbol{Z})} q(\boldsymbol{Z}) d\boldsymbol{Z} && \text{multiplying by 1 inside} \\
&= \frac{1}{N} \log \int \frac{p(\boldsymbol{X}, \boldsymbol{Z})}{q(\boldsymbol{Z})} dq(\boldsymbol{Z}) && \text{definition of } dq(\mathbf{Z}) \\
&\geq \frac{1}{N} \int \log \frac{p(\boldsymbol{X}, \boldsymbol{Z})}{q(\boldsymbol{Z})} dq(\boldsymbol{Z}) && \text{Jensen inequality} \\
&= \frac{1}{N} \int \sum_{1}^{N} \log \frac{p(\mathbf{x}_i, \mathbf{z}_i)}{q(\mathbf{z}_i)} dq(\mathbf{z}_i) && \text{using the iid property 3.1} \\
&= \frac{1}{N} \sum_{1}^{N} -\mathcal{L}(q, p, \mathbf{x}_i) && \text{definition of } \mathcal{L}(q, p, \mathbf{x}_i) \\
&= -\mathcal{L}(q, p, X) \triangleq -\mathcal{L}(q, p) && \text{again definition of } \mathcal{L}(p, q) \quad \square
\end{aligned} \tag{3.4}$$

In equation 3.4 we found a lower bound $-\mathcal{L}(q, p)$ for the mean log evidence $\log p(\boldsymbol{X})/N$, the *ELBO*. Whatever distribution q we put in ELBO will not be greater than the real log evidence so we are looking for the q which **maximizes** it.

Now we show that maximizing the ELBO actually obtains the mean log evidence and it is equivalent to minimizing $KL(q(\boldsymbol{Z}) \| p(\boldsymbol{Z}|\boldsymbol{X}))$:

$$
\begin{aligned}
-\mathcal{L}(q,p,\mathbf{x}) &\triangleq \int \log \frac{p(\boldsymbol{x},\boldsymbol{z})}{q(\boldsymbol{z})} dq(\boldsymbol{z}) = \int \log \frac{p(\boldsymbol{z}|\boldsymbol{x})p(\boldsymbol{x})}{q(\boldsymbol{z})} dq(\boldsymbol{z}) \\
&= \int \log p(\boldsymbol{x}) dq(\boldsymbol{z}) - \int \log \frac{q(\boldsymbol{z})}{p(\boldsymbol{z}|\boldsymbol{x})} dq(\boldsymbol{z}) = \log p(\boldsymbol{x}) - KL(q(\boldsymbol{z})\|p(\boldsymbol{z}|\boldsymbol{x}))
\end{aligned} \tag{3.5}
$$

We can rewrite equation 3.5 as:

$$
\log p(\boldsymbol{x}) = -\mathcal{L}(q,p,\mathbf{x}) - KL(q(\boldsymbol{z})\|p(\boldsymbol{z}|\boldsymbol{x})) \tag{3.6}
$$

Equation 3.6 shows that the ELBO minus the KL-divergence are constant and equal the log evidence. Therefore minimizing the KL-divergence (which is always non-negative) simultaneously maximizes the ELBO and vicer-versa.

3.2 Variational Autoencoder

3.2.1 Adding parameters

Our models will not be fully Bayesian, but rather parametrized. Suppose that the p distribution over $\mathbf{x}, \mathbf{z}$ belongs to some parametrized family of distributions $p_\theta(\mathbf{x})$ and the q distribution over $\mathbf{z}$ belongs to another family $q_\phi(\mathbf{z})$. In a fully Bayesian model we would make θ and ϕ stochastic parameters and give them appropriate prior distributions, but with the VAE we leave them as parameters that we determine with neural network as will shortly be explained.

For any θ and any ϕ, the equations from the previous chapter hold also in the parametrize form, i.e $\log p_\theta(\mathbf{x}) = -\mathcal{L}(q_\phi, p_\theta, \mathbf{x}) - KL(q_\phi(\mathbf{z})\|p_\theta(\mathbf{z}|\mathbf{x}))$.

We assume that we can only approach the "real" distribution using θ from below $\log p(\mathbf{x}) \geq \log p_\theta(\mathbf{x})$. So together with equation 3.4 we have

$$
(\forall\theta,\phi) \log p(\mathbf{x}) \geq \log p_\theta(\mathbf{x}) \geq -\mathcal{L}(q_\phi.p_\theta,\mathbf{x}) = \int \frac{p_\theta(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z})} dq_\phi(\mathbf{z}) \tag{3.7}
$$

And from equation 3.6 we again see that by finding the parameters ϕ, θ that maximize the ELBO we approach the real log evidence as much as we can within the limits of the parametrized family of distributions we use.

3.2.2 Rearranging the ELBO

Equations 3.4 and 3.5 were defined for any distribution $q(\mathbf{z})$ and in particular we are allowed to plug in a conditioned distribution $q(\mathbf{z}|\mathbf{x})$. That implies the existence of $q(\mathbf{z},\mathbf{x})$ and $q(\mathbf{x})$ but we actually don't care about them. We condition everything on $\mathbf{x}$ but $\mathbf{x}$ is treated as a given constant from a Bayesian view point and we only want to somehow make $q(\mathbf{z}|\mathbf{x})$ to closely approximate $p(\mathbf{z}|\mathbf{x})$.

A second thing we need to achieve is to express the -ELBO in terms of $p(\mathbf{x}|\mathbf{z})$ and $q(\mathbf{z}|\mathbf{x})$ rather than the joint distribution. To that end we need also the prior $p(\mathbf{z})$.

$$\begin{aligned}\mathcal{L}(q,p,\mathbf{x}) &\triangleq \int -\log\frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z}|\mathbf{x})}dq(\mathbf{z}|\mathbf{x}) = \int -\log\frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{q(\mathbf{z}|\mathbf{x})}dq(\mathbf{z}|\mathbf{x}) \\ &= \int -\log p(\mathbf{x}|\mathbf{z})dq(\mathbf{z}|\mathbf{x}) + \int \log\frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})}dq(\mathbf{z}|\mathbf{x}) \\ &= \int -\log p(\mathbf{x}|\mathbf{z})dq(\mathbf{z}|\mathbf{x}) + KL(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}))\end{aligned} \tag{3.8}$$

So to sum it up, if we want to maximize the log evidence $\log p(\mathbf{X})$ it suffices to minimize $\mathcal{L}(q,p)$ and equation 3.8 shows that this means finding the balance between making the term $\int \log p(\mathbf{X}|\mathbf{Z})dq(\mathbf{Z}|\mathbf{X})$ (which we call the reconstruction term) large as possible, and making the KL-term small. The KL term is seen as a regularization term.

3.2.3 Using neural networks for the parametrization

In this text we deal with variational autoencoders (VAE). A VAE is a neural network which is used to define and optimize the parameters ϕ and θ which define $p_\theta(\mathbf{x}|\mathbf{z})$ and $q_\phi(\mathbf{z}|\mathbf{x})$ and we train the network to maximize equation 3.8.

Specifically the encoder part of the network is a feed forward neural network $f_\theta(\mathbf{z})$ which is used to define the distribution $p_\theta(x|z)$. For example, we can assume that p_θ is a family of multivariate Gaussians and in this case $f_\theta(z) = (\mu(z), \Sigma(z))$. Meaning the encoder maps z to the location vector and covariance matrix. The parameter θ in this case are the weights of the encoder neural network.

The decoder network is similarly defined as neural network $g_\phi(\mathbf{x})$ which maps $\mathbf{x}$ into the parameters defining the family $q_\phi(\mathbf{z})$. Here too ϕ represent the weights of the decoder.

For the prior $p(\mathbf{z})$ we set some fixed prior distribution.

Note that the encoder network (similarly the decoder) is used to define a distribution over $\mathbf{z} \in \mathbb{R}^m$, but the encoder itself maps into some other space. For example, to define a normal one Gaussian distribution (so over $\mathbf{z} \in \mathbb{R}$, the encoder maps into $\mathbb{R}^2$, and its output creates $\mu, \sigma\mathbb{R}$ which are used to define the Gaussian distribution $\mathcal{N}(\mathbf{z}; \mu, \sigma)$. We also need to make sure that the range of the network obeys to the constraints of the parameters. For example the variance must be non-negative. Alternatively we can use transformations to remove constraints. For example instead of letting the network specify the variance. we let it specify the log-variance.

Definition 3.3. Let $\{p_\theta\}_{\theta\in\Theta}$ be a parameterized family of distributions over $\mathbb{R}^n$ and let $\{q_\phi\}_{\phi\in\Phi}$ be a family of distributions over $\mathbb{R}^m$. Where Θ and Φ are real domains (i.e $\Theta \subseteq \mathbb{R}^k \dots$).

A *variational autoencoder (VAE)* consists of a pair (E, D) of neural networks, $E : \mathbb{R}^n \to \Phi$ and $D : \mathbb{R}^m \to \Theta$ and some fixed distribution $p \in \Phi$.

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

The definitions given for Autoencoder and for VAE are independent. We don't require for one to be a subtype of the other. However as the name suggests these two are connected. An autoencoder works deterministically, where the encoder maps the input $\mathbf{x} \mapsto \mathbf{z}$ and the decoder then maps the latent space $\mathbf{z} \mapsto \hat{\mathbf{x}}$ to the reconstruction. A VAE does basically the same thing but non-deterministically. It maps $\mathbf{x}$ into a distribution over $\mathbf{z}$: $\mathbf{x} \mapsto q(\mathbf{z}|\mathbf{x}) = q_{\phi(\mathbf{x})}(\mathbf{z})$ and it maps $\mathbf{z}$ into a distribution over $\mathbf{x}$: $\mathbf{z} \mapsto p(\mathbf{x}|\mathbf{z}) = p_{\theta(\mathbf{z})}(\mathbf{x})$.

The loss function associated with a VAE is minus ELBO. This means training the VAE maximizes equation 3.8 and therefore also the log evidence.

To give a concrete example lets say that the distribution family for q_ϕ is the diagonal normal distribution. We can set the parameter domains $\Phi = \mathbb{R}^m \oplus \mathbb{R}^m$. The means can be any real vector, but the variances are non-negative. We can either restrict the encoder to be non-negative on the variance-domain, or we can agree to use the log-variance as the second parameter, which is then unconstrained.

We identify $q(\mathbf{z}|\mathbf{x})$ with the encoder but technically what it means is that the encoder network maps $\mathbf{x}$ into the distribution parameters $E(\mathbf{x}) = \phi(\mathbf{x}) \in \Phi$; in the case we use diagonal normal distribution $\phi(\mathbf{x}) = (\mu(\mathbf{x}), \sigma(\mathbf{x}))$. Thus we defined a distribution over $\mathbf{z}$ by using $\mathbf{x}$ to map into the parameter domain: $q(\mathbf{z}|\mathbf{x}) = q_{\phi(\mathbf{x})}(\mathbf{z})$

Similarly we call $p(\mathbf{x}|\mathbf{z})$ the decoder although it is technically the distribution defined by the decoder. $D(\mathbf{z}) = \theta(\mathbf{z})$: $p(\mathbf{x}|\mathbf{z}) = p_\theta(\mathbf{x})$

3.2.4 Mean field approximation

Usually we treat the dimensions of $\mathbf{x}$, $\mathbf{z}$ etc. as independent. That means if $\mathbf{x} = (x_1, \ldots, x_n)$ is a random vector. in $\mathbb{R}^n$ we assume that the x_i are independent and therefore $P(\mathbf{x}) = \prod_1^n p_i(x_i)$.

Specifically the mean fields approximation of a multivariate Gaussian $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is a diagonal Gaussian distribution $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\sigma}) \triangleq \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\sigma}\boldsymbol{I})$. Mean field approximation simplifies the implementation and speeds up the computation and has been the standard practice since the beginning of VAEs [9].

3.2.5 Computing the ELBO

It may not be immediately clear how to how to compute the integral in the ELBO function. Recall that given an input $\mathbf{x} \in \mathbb{R}^n$, The loss function is

$$\mathcal{L}(p, q, \mathbf{x}) = \int -\log \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{q(\mathbf{z}|\mathbf{x})} dq(\mathbf{z}|\mathbf{x}) = \int -\log p(\mathbf{x}|\mathbf{z}) dq(\mathbf{z}|\mathbf{x}) + \int \log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})} dq(\mathbf{z}|\mathbf{x}) \tag{3.9}$$

Given concrete input $\mathbf{x} \in \mathbb{R}^n$, the decoder specifies a distribution over $\mathbf{z} \in \mathbb{R}^m$ rather then a concrete deterministic point: $\mathbf{z} \sim q(\mathbf{z}|\mathbf{x})$. Suppose that we draw one concrete sample $\mathbf{z} \in \mathbb{R}^m$ taken from that distribution. Now that we have the concrete input $\mathbf{x}$ and a concrete $\mathbf{z}$ we can compute $\log q(\mathbf{z}|\mathbf{x})$ as well as the prior $p(z)$. Remember that the decoder takes $\mathbf{z}$ and produces a distribution $p(\mathbf{x}|\mathbf{z})$. With a concrete $\mathbf{z}$, and $\mathbf{x}$ we can also

compute $\log p(\mathbf{x}|\mathbf{z})$ So once we draw a specific sample $\mathbf{z}$ we can compute everything inside the integral.

In fact what we have done is already a form of Monte Carlo integration. More generally, instead of drawing just one concrete sample $\mathbf{z}$, we draw k samples $\mathbf{z}_i \sim q(\mathbf{z}|\mathbf{x})$ per input $\mathbf{x}$, and take the average. Then we have

$$\begin{aligned} \mathcal{L}(p, q, \mathbf{x}) &= \int -\log \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{q(\mathbf{z}|\mathbf{x})} dq(\mathbf{z}|\mathbf{x}) \\ &= \int -\log p(\mathbf{x}|\mathbf{z}) dq(\mathbf{z}|\mathbf{x}) + \int \log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})} dq(\mathbf{z}|\mathbf{x}) \\ &\approx \frac{1}{k} \sum_{i=1}^{k} [-\log p(\mathbf{x}|\mathbf{z}_i) + \log \frac{q(\mathbf{z}_i|\mathbf{x})}{p(\mathbf{z}_i)}] \end{aligned} \tag{3.10}$$

In practice we take just one ($k = 1$) sample z for each input data point x. Remember that we are working on batches and computing an average loss over batches so for a given batch we are taking many samples $\mathbf{z}$. Experimental data shows that taking larger samples usually brings little benefit [8].

Reparameterization trick

The loss function is computed by using Monte Carlo integration, which requires taking samples $\mathbf{z} \sim q(\mathbf{z}|\mathbf{z})$. But the loss function needs to be differentiable with respect to the model's parameters which are the encoder's weights ϕ and the decoder's weights θ. There is a method, "the reparameterization trick" which uses some random noise which allows to express the sample $\mathbf{z} = h_\phi(\epsilon, \mathbf{x})$ as a deterministic smooth function of the parameters given random noise ϵ [9].

For example in the "vanilla" VAE, $\mathbf{z}$ is sampled from diagonal Gaussian distribution $\phi(\mathbf{x}) = (\mu(\mathbf{x}), \sigma(\mathbf{x}))$. Sampling $\mathbf{z} \sim \mathcal{N}(\mu(\mathbf{x}), \sigma(\mathbf{x}))$ is equivalent to sampling standard normal noise $\epsilon \sim \mathcal{N}(0, 1)$ and taking $\mathbf{z} = \mu(\mathbf{x}) + \sigma(\mathbf{x}) \cdot \epsilon$. Most other types of distributions (Dirichlet, Negative Binomial, etc.) can be sampled using the reparameterization trick as deterministic, differentiable transformation of Gaussian or uniform random noise and in Pytorch, this feature is built into the distribution object types.

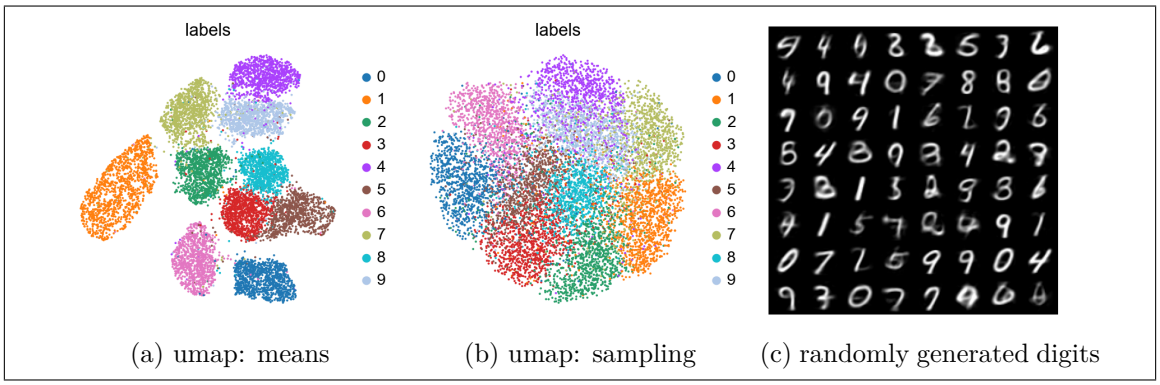
3.2.6 Using the decoder for data generation

Suppose that we have a vanilla auto encoder (ϕ, ψ) and suppose that we want to generate synthetic data set that looks similar to the original data. We need to choose points $\mathbf{z} \in \mathbb{R}^m$ from the latent space and project back to observed space $\psi(\mathbf{z}) \in \mathbb{R}^n$. The question is then how to sample these $\mathbf{z}$? It's not immediately clear how to do this because we don't know what is the distribution in the latent space.

If we use a VAE instead, we know that $\mathbf{z}$ should have a distribution that is pretty close to the prior $p(\mathbf{z})$, which we can easily sample from. Given a VAE (E, D, p), synthetic data samples can be generated as follows: sample $\mathbf{z} \sim p(\mathbf{z})$. Then given the samples in the latent space, sample from the decoder distribution $\mathbf{x} \sim D(\mathbf{z})$, in the observed space. Figure 3.1c shows randomly generated digits which were created in such process.

3.2.7 Encoding data in the latest space with a VAE

Unlike a vanilla autoencoder, a VAE doesn't deterministically encode input $\mathbf{x}$ but rather maps it to a distribution $q(\mathbf{z}|\mathbf{x})$. We can use the distribution's mean $\mu(\mathbf{x})$ as the encoding. Given observation $\mathbf{x}$, we can deterministically encode $\mathbf{x}$ into the latent space by taking the mean: $\mathbf{x} \mapsto \mathbf{E}[E(\mathbf{x})]$. Since $q(\mathbf{x}|\mathbf{z})$ is some kind of parameterized distribution for example $\mathcal{N}(\mathbf{z} : \mu(\mathbf{x}), \sigma(\mathbf{x}))$ the mean is a known parameter of the distribution so we don't need to estimate it. See for example in figure 3.1a. Typically encoding by the means looks more like an encoding with a deterministic autoencoder would look. Alternatively we can deliberately add stochasticity to the encoding by non-deterministically drawing $\mathbf{z} \sim q(\mathbf{z}|\mathbf{x})$ from the encoder distribution, as is dones in figure 3.1b. When we use this sampling technique for the encoding, we see how the VAE approximates the standard normal prior distribution in the latent space.



(a) umap: means (b) umap: sampling (c) randomly generated digits

Figure 3.1: "vanilla" VAE trained on MNIST. The first two images show UMAP plot of the latent space $\mathbf{z}$. In (a) we take the mean of the distribution $p(\mathbf{z}|\mathbf{x})$ while in (b) a random sample from the distribution is used. Third plot shows random digits generated by sampling $z \sim p(\mathbf{z}|\mathbf{x})$ and projecting back to the observed space $\mathbf{x}$ by the decoder.

3.2.8 Choosing the distribution types

Recall that our loss function in the case that we take just one sample $\mathbf{z}$ for input $\mathbf{x}$ is:

$$\begin{aligned} \mathcal{L}(p, q, \mathbf{x}) &= (-\int \log p(\mathbf{x}|\mathbf{z}) dq) + KL(q(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z})) \\ &\approx -\log p(\mathbf{x}|\mathbf{z}) + \log \frac{\log q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})} \end{aligned} \tag{3.11}$$

The left term is called the reconstruction error and the right term is called the regularization term or the kl-term.

Now suppose that we just want to use diagonal Gaussian distributions (which is a type of mean field approximation). The advantage is that it is easy to compute them because we just sum over the dimensions:

$$\log \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\sigma}) = \sum_{1}^{n} \log \mathcal{N}(x_i; \mu_i, \sigma_i) \tag{3.12}$$

The dimension of the latent space is a significant hyper parameter of the VAE model.

Since we sum over the dimensions rather than averaging, the larger we let the dimension of the latent space $\mathbf{z}$ it can have an effect of upscaling the importance of the kl-term relative to the reconstruction. Moreover the dimension should also be appropriate in terms of the "real" dimensionality of the data.

In the case of the vanilla VAE we choose $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}; 0, 1)$ (diagonal Gaussian standard normal) for the prior and $q(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}(\mathbf{x}), \boldsymbol{\sigma}(\mathbf{x}))$ for the encoder. There is a closed form formula for KL-divergence between two diagonal Gaussians so in this case we don't need to use Monte Carlo integration for the KL-term (we show it in one dimensions and for k dimension in the diagonal case we sum over the dimensions:

$$KL(\mathcal{N}(;\mu_1,\sigma_1)\|\mathcal{N}(;\mu_2,\sigma_2)) = -\frac{1}{2} + \log\frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} \tag{3.13}$$

Choosing the right type of distribution $p(\mathbf{x}|\mathbf{z})$ for the reconstruction term $-\log p(\mathbf{x}|\mathbf{z})$ is also important. Lets say we still want some sort of Gaussian diagonal distribution for $\mathbf{x}$, so $p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}(\mathbf{z}), \boldsymbol{\sigma}(\mathbf{z}))$. If we allow the decoder to create arbitrarily small variances, then the Monte Carlo integration, at least the simple one we use, fails. The reason is that the decoder make the variance infinitesimally close to 0 and then the density function becomes essentially unbounded from below. The error term becomes negative and is dominated by the negative reconstruction term and the model might also become numerically unstable in training because the loss diverges to $-\infty$.

A common approach to avoid this problem is to use a fixed variance of 1, because then the reconstruction term becomes square error loss. $p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}(\mathbf{z}), 1))$. Another method, σ-VAE [22], is to let the variance be a trainable parameter of the model, but not a function of $\mathbf{z}$, rather it is is shared by all the samples. $p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}(\mathbf{z}), \boldsymbol{\sigma})$. In this case as well, in our experiments at least, if σ is allowed to be too small we sometimes get similar issues of unbounded negative loss and numerical instability.

In the case that the data $\mathbf{x}$ contains only values in the range $[0, 1]$, which is the case for example for images, it is common to choose Bernoulli distribution to model the reconstruction distribution. Every pixel is interpreted as a Bernoulli random variable with probability equal to the measure intensity. The loss function for the reconstruction term becomes binary cross–entropy.

3.2.9 VAE as a generalization of AE

Suppose that we take a σ-VAE as described above, and suppose that we hold σ fixed.

The decoded distribution is $p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}; \mu(\mathbf{z}), \sigma)$ where $\mu(z)$ is a function of $\mathbf{z}$ the decoder's neural network. The encoded distribution is $q(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \mu(\mathbf{x}), \sigma(\mathbf{x}))$. For the kl-term we use the analytical solution and for the reconstruction error we use Monte Carlo integration with one sample. We can assume that the KL term is uniformly (with respect to $\mathbf{x}$) bounded by some constant M. We could also make sure that the encoder map is bounded but normally there is no reason for the encoder to take the mean or the variance to infinity during training as it would just increase the loss.

Our -ELBO loss function is:

$$\begin{aligned}\mathcal{L}(p,q,\mathbf{x}) &= -\log\mathcal{N}(\mathbf{x};\mu(\mathbf{z}),\sigma) + KL(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})) \\ &= \frac{1}{2}\|\frac{x-\mu(\mathbf{z})}{\sigma}\|^2 + \log\sigma + KL(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}))\end{aligned} \tag{3.14}$$

Minimizing $\mathcal{L}(p,q,\mathbf{x})$ in equation 3.14 is equivalent to minimizing $\sigma\mathcal{L}(p,q,\mathbf{x})$, and if we let $\sigma \to 0$ we get:

$$\begin{aligned}\lim_{\sigma\to 0}\mathcal{L}(p,q,\mathbf{x})\sigma &= \lim_{\sigma\to 0}[-\log\mathcal{N}(\mathbf{x};\mu(\mathbf{z}),\sigma)\sigma + \sigma KL(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}))] \\ = \lim_{\sigma\to 0}[\frac{1}{2}\|x-\mu(\mathbf{z})\|^2 + \sigma\log\sigma + \sigma KL(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}))] &= \frac{1}{2}\|x-\mu(\mathbf{z})\|^2\end{aligned} \tag{3.15}$$

So if we set σ be arbitrary small, $p(\mathbf{x}|\mathbf{z})$ become almost point mass and the KL term loses significance, effectively removing the constraint from the encoder. As a result the encoder itself will also become point mass in order to minimize the reconstruction term and disregarding the constraint on its distribution. Thus the VAE becomes essentially a vanilla auto encoder at the limit.

3.2.10 Graphical representation

It is both convenient as well as informative to include a graphical description of our probabilistic models. To that end we use a modified plate diagram which describes both the generative model (p distribution) and the inference model (q) on the same graph.

Definition 3.4. A *modified plate diagram* describes the factorization of two probabilistic models with respectively distributions p and q, according to the following scheme.

Round or oval nodes represent *random variables*, *rectangular or square* nodes represent *hyperparameters* or parameters, which are used for specifying prior distributions. *Arrows* represent dependency relation.

Solid arrows with pointed arrowhead represent the dependencies of the p distribution. *Dotted arrows with round arrowheads* represent the dependencies of the q distribution. In order to describe a legal distribution the subgraph of just the pointed (and of just round) arrows must form a *DAG* (=directed acyclic graph). A *Plate* Is a rectangular frame which encapsulates one or more round/oval nodes. It represents the packing of N i.i.d random vectors. Specifically in all the diagrams we will show here, N represents the number of observations in the dataset.

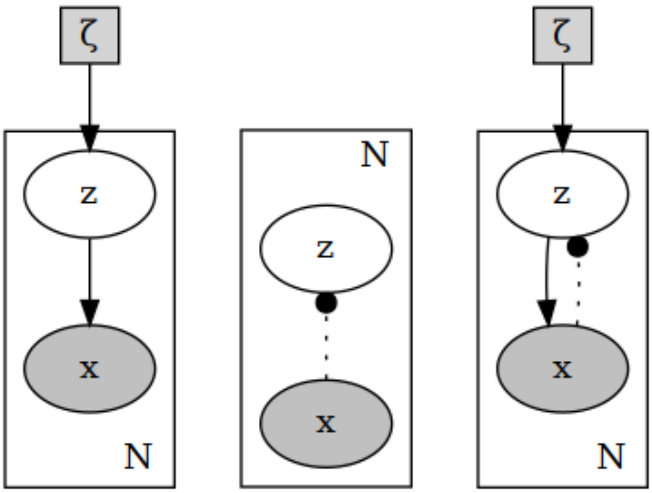
Shaded nodes represent known values or observations. Shaded oval or round nodes are called *observations* or *observed variables*. Clear oval nodes are called *latent* variables. Shaded square or rectangular nodes represent hyperparameters of the model. We need them to specify prior distributions of the generative model. It is possible to drop the shaded square nodes and just assume that every clear round/oval node in the DAG which is a source (no incoming solid arrows or no incoming dotted arrows) implicitly has some predefined prior distribution. In this thesis we don't use clear square or rectangular nodes. In principle those can be used to represent learnable non-stochastic parameters of the model.

Figure 3.2 is a modified plate diagram of the VAE model. We use doted arrows with round arrowhead to represent the inference model (encoder network), and regular arrows

for the generative model (decoder network) so the combined diagram describes the two networks together.

The squared ζ node represent some *fixed hyperparameter* which describes the prior distribution of $p(\mathbf{z}) := p(\mathbf{z}|\zeta)$.

The generative model therefore factors as: $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z}|\zeta) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z})$. The inference model in this case is just $q(\mathbf{z}|\mathbf{x})$.



(a) generative model ($p(x, z)$) (b) inference model ($q(z|x)$) (c) the combined graphical model

Figure 3.2: VAE graphical model

Note that the graphical model has no assumption about the specific types of distributions involved (Gaussian, Dirichlet or whatever …) and that is left for the actual implementation.

In the case of a "vanilla" VAE (E, D, p), Mean field approximation, with diagonal Gaussian distributions for p and q, is used. The prior $p(\mathbf{z})$ is set to be diagonal standard Gaussian $p(\mathbf{z}) \sim \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{1})$. $p(\mathbf{z}|\mathbf{x}) \sim \mathcal{N}(\mathbf{z}; E(\mathbf{x}))$ is a diagonal Gaussian, where the encoder neural network $E(\mathbf{x})$ determines its means and variances $E(\mathbf{x}) = (\boldsymbol{\mu}(\mathbf{x}), \boldsymbol{\sigma}(\mathbf{x}))$, And similarly with the decoder $p(\mathbf{x}|\mathbf{z}) \sim \mathcal{N}(\mathbf{x}; D(\mathbf{z}))$.

3.3 Expanding the VAE model

If we look at figure 3.2 it looks very simple, but it also pretty much forces us to choose a simple type of distribution family (e.g diagonal Gaussians in the case of the vanilla VAE). Recall that That $\mathbf{z}$ packs up all the latent variables and the stochastic parameters and $\mathbf{x}$ packs up all the observed variables.

We can describe a more complex distribution by unpacking them and describe the dependencies between them. This is done in the following way:

1. Define the set of observed random vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_k$, and the set of latent random vectors and stochastic parameters $\mathbf{z}_1, \ldots \mathbf{z}_l$.

2. Specify how to factor the generative model $p(\mathbf{x}_1, \ldots, \mathbf{x}_k|\mathbf{z}_1 \ldots, \mathbf{z}_l)$
3. Specify how to factor the inference model $q(\mathbf{z}_1 \ldots \mathbf{z}_l|\mathbf{x}1, \ldots \mathbf{x}_k)$
4. Choose appropriate priors $p(\mathbf{z}_i)$ and
5. Choose appropriate distribution families for the $\mathbf{x}_i$ and $\mathbf{z}_i$, and choose priors $p(\mathbf{z}_i)$.

3.3.1 Example: CVAE

Suppose that we have data that carries both numerical and categorical data $(\mathbf{X}, C)$. For example suppose that $\mathbf{X}$ represent a set of images (as flattened vectors), and C represents the object types shown in the images. Moreover lets assume that we have k types of categories and that the data is balanced so we have k/N samples from each category. We have just specified our observed variables $\mathbf{x}, c$. We will have one latent variable $\mathbf{z}$ (remember it is actually a vector but we call them variables...). The idea here is that because we have different categories, we will have some type of a mixture of distributions.

Lets specify the generative model $p(\mathbf{x}, \mathbf{z}, c)$. We can factor it "arbitrarily" however the choice may make a significant difference on the result. In this case there aren't too many ways to factor. We can factor as $p(\mathbf{x}, \mathbf{z}, c) = p(\mathbf{x}|\mathbf{z}, c)p(\mathbf{z}|c)p(c)$ which mean $\mathbf{x}$ is directly dependent on both $\mathbf{z}$ and c. If we use this factorization then we concatenate $\mathbf{z}$ and c for computing the decoder's reconstruction $p(\mathbf{x}|\mathbf{z}, c)$. But suppose we assume that $\mathbf{x}$ and c are conditionally independent given $\mathbf{z}$, so we may set $p(\mathbf{x}, \mathbf{z}, c) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z}|c)p(c)$. We call $p(\mathbf{z}|c)$ a "learned prior" of $\mathbf{z}$ since it is not fixed like in the VAE case but rather the decoder maps the categorical c into some distribution.

As for the inference model (encoder), given the observation $\mathbf{x}$ and c it will determine our only latent variable $\mathbf{z}$, in other words $q(\mathbf{z}|\mathbf{x}, c)$ is the inference model without anything further to factorize. Conditioning in c is done here as well by concatenating the inputs $\mathbf{x}$ and c.

Since our data is balanced, we use uniform prior $p(c) = \frac{1}{k}$.

The generative process (decoder) is therefore as follows:

- draw a category $c \sim Cat(\frac{1}{k})$.
- draw $\mathbf{z} \sim p(\mathbf{z}|c)$.
- draw $\mathbf{x} \sim p(\mathbf{x}|\mathbf{z})$.
- the resulting factorization of p is: $p(\mathbf{x}, \mathbf{z}, c) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z}|c)\frac{1}{k}$.

Remember that the loss function is still the minus ELBO, which according to our fac-

torization becomes:

$$\begin{aligned}\mathcal{L}(p,q,\mathbf{x},c) &= \int -\log \frac{p(\mathbf{x},c,\mathbf{z})}{q(\mathbf{z}|\mathbf{x},c)} dq \\ &= \int -\log \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z}|c)p(c)}{q(\mathbf{z}|\mathbf{x},c)} dq \\ &= \int -\log p(\mathbf{x}|\mathbf{z}) dq + \int \log \frac{q(\mathbf{z}|\mathbf{x},c)}{p(\mathbf{z}|c)} dq + \log(k) \\ &= \int -\log p(\mathbf{x}|\mathbf{z}) dq + KL(q(\mathbf{z}|\mathbf{x},c)\|p(\mathbf{z}|c)) + \text{const}\end{aligned} \tag{3.16}$$

Since the $\mathbf{z}$ prior depends on the category, $p(\mathbf{z}|c)$, it should be some sort of "blobs" mixture type of distribution. The inference model is just $q(\mathbf{z}|\mathbf{x},c)$. The regularization kl-term tries to impose $q(\mathbf{z}|\mathbf{x},c)$ to be close to $p(\mathbf{z}|c)$, so if all works well $q(\mathbf{z}|\mathbf{x},c)$ should look like a mixture distribution ("blobs").

Now for concrete choice of distribution families: $p(c)$ is already chosen for us as uniform categorical. For the rest we again use diagonal Gaussians. $p(\mathbf{z}|c)$ will be parametrized by an encoder network taking only the categorical information. Essentially this network will map each category into some "blob" around some centroid in the latent space. $p(\mathbf{x}|\mathbf{z})$ describes how given $\mathbf{z}$ it defines a diagonal normal distribution with fixed (or restricted) variance back in the observed space like the decoder network in the vanilla case. $q(\mathbf{z}|\mathbf{x},c)$ means that in this case the encoder takes as input both $\mathbf{x}$ and c and defines a diagonal Gaussian in the latent space. The difference is that with this model after we train it, the encoder will encode a mixture distribution in $\mathbf{z}$, we will get several blobs in the latent space corresponding to the classes.

From equation 3.16, we can ignore the constant and see that the reconstruction term that will make sure the decoder reconstruct the image in our example, while the kl-term imposes a mixture distribution in the latent space.

Finally there are circumstances that we use CVAE to "forget" the categories rather then to encode them by setting a fixed prior $p(\mathbf{z}|c) \equiv p(\mathbf{z})$. An example for such use-case is for batch effect reduction. In cases where for example, there are several batches of data of the same type and but from different experiments. We expect any differences in the data of the same entity are a result of technical differences (different measuring tools etc.) rather than reflecting true differences between the entity of interest. In this case we can use a CVAE model with fixed prior to reduce the batch effect.

Figure 3.4 shows in the left a UMAP plot of the latent space for the CVAE model which we described first, on the MNIST data set. It uses the learned prior $p(\mathbf{z}|c)$ and $\mathbf{x}$ and c are conditionally independent given $\mathbf{z}$. The middle image shows the reconstruction of the clusters centers (the means of $p(\mathbf{z}|c)$). In this model the encoder completely separates the categories in the latent representation, resulting in distinct blobs. The right image is the CVAE described last, the one which "forgets" the category, with fixed prior $p(\mathbf{z}|c) = p(\mathbf{z}) = \mathcal{N}(\mathbf{z}; 0, 1)$. In this model $\mathbf{x}$ is not conditionally independent from c, but instead it is directly dependent on both $\mathbf{z}$ and c ($p(\mathbf{x}|\mathbf{z},c)$). The result is an encoding which mixes a lot of the categories in the latent space but we can still notice some clustering.

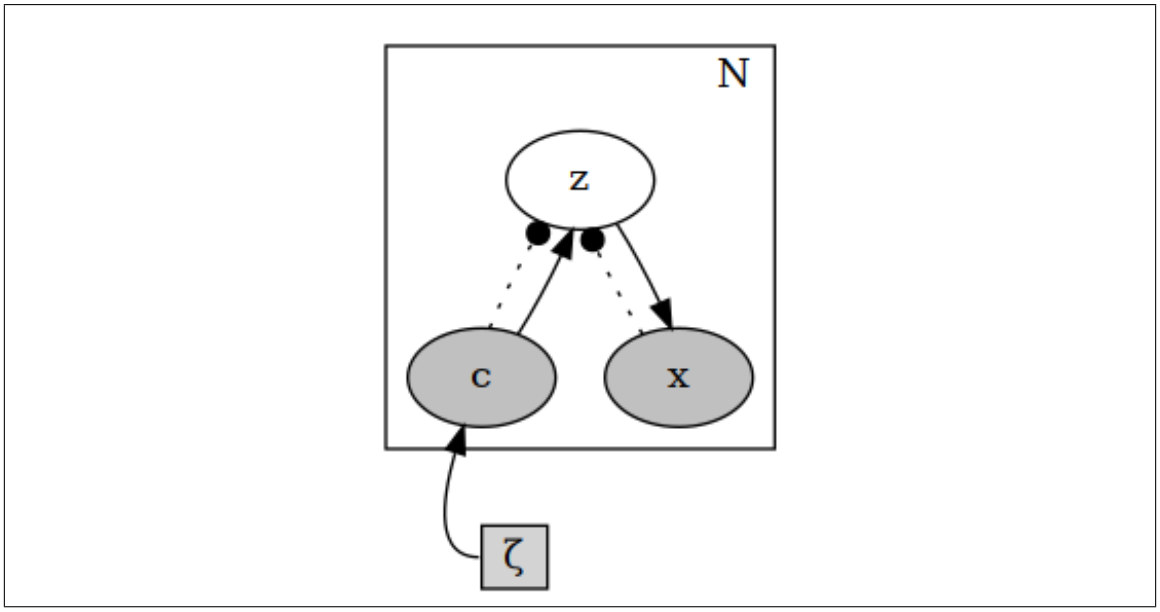
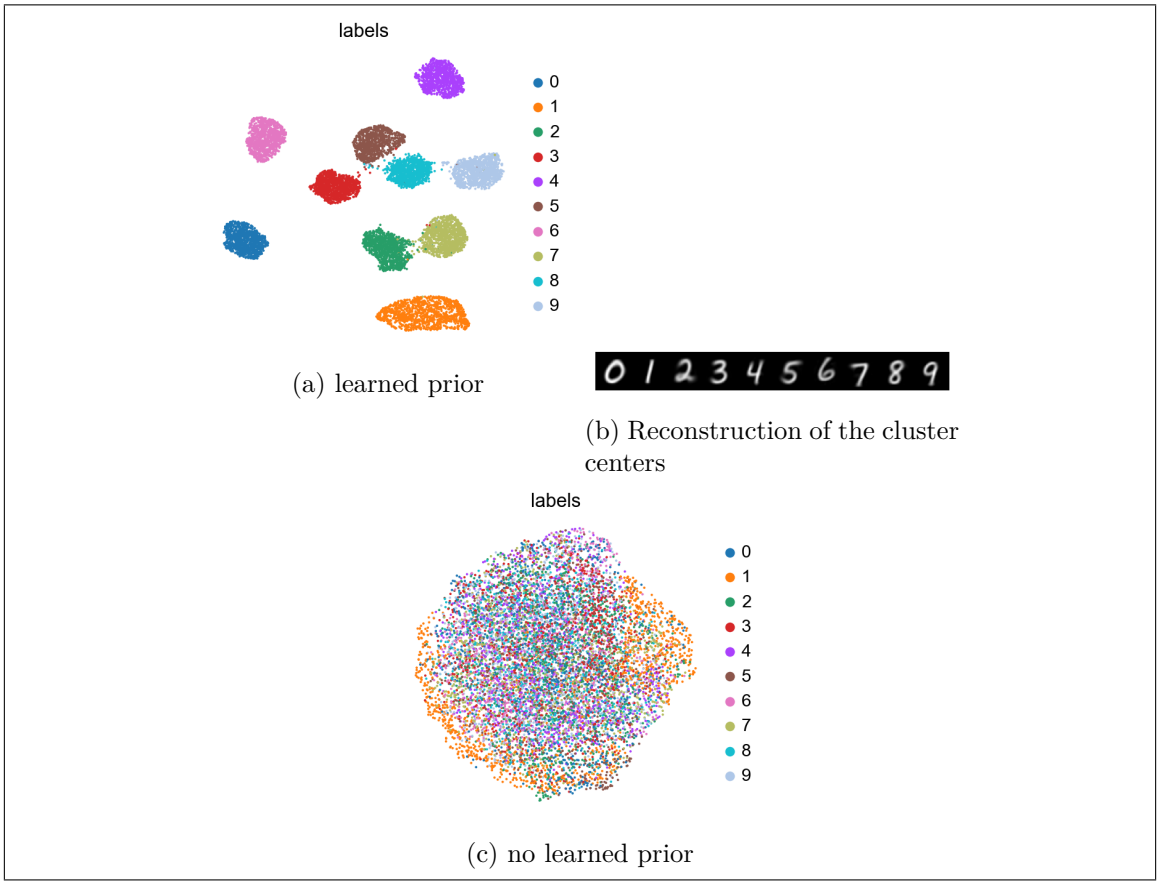


Figure 3.3: Graphical model of the CVAE with a learned prior $p(\mathbf{z}|c)$. as usual the solid arrows depict the inference model (encoder) and the dotted ones the generative model (decoder)



(a) learned prior

(b) Reconstruction of the cluster centers

(c) no learned prior

Figure 3.4: CVAE, different use cases

Chapter 4

Gaussian mixture model VAEs

4.1 Motivation

Suppose that we have data of type $(\mathbf{x}, \mathbf{y})$ which like in the CVAE case has numerical and categorical information, but unlike in the CVAE section we don't have access to the categories $\mathbf{y}$, only to $\mathbf{x}$. We are looking for a model that can partition $\mathbf{X}$ according to the true categories. This is a task of unsupervised learning which is much harder then classifying data with known targets $\mathbf{Y}$.

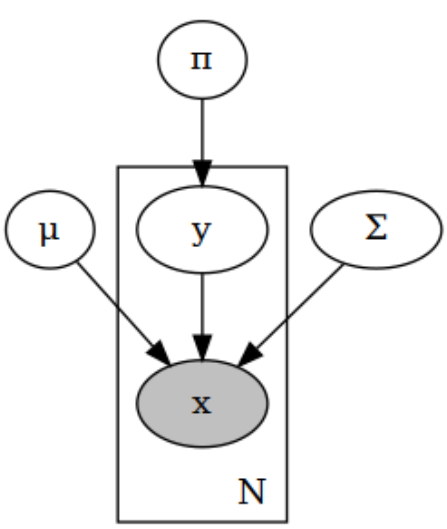
The original data has a complex distribution. Not only is it high dimensional, but because the data belongs to distinct categories, it is natural to assume that $\mathbf{x}$ comes from a mixture distribution.

We want to create some mixture distribution, where a latent categorical random variable $\mathbf{y}$ functions as the component selector. We could try to directly create the mixture $p(\mathbf{x}|\mathbf{z}, \mathbf{y})$ in the $\mathbf{x}$ space, as the M2 model in [10]. The other option, following the Gaussian mixture VAE model, *GMVAE*, proposed by Dilokthanakul et.al [2] and on which we base our c∗GMΔVÆ model, is to create a mixture prior distribution $p(\mathbf{z}|\mathbf{w}, \mathbf{y})$ on the lower dimensional latent $\mathbf{z}$ space.

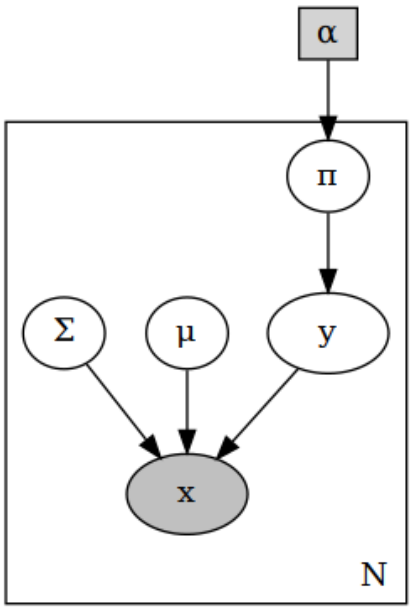
There are several advantages to doing this. For once we are interested in dimensionality reduction and not just in unsupervised classification. If we create a mixture distribution in the latent space $\mathbf{z}$ we can then further analyse it (by clustering algorithms for example) and plot it, for example with UMAP. A second reason is that it may be easier to create a mixture Gaussian on the simpler, lower dimensional space $\mathbf{z}$ rather than on the high dimensional $\mathbf{x}$ space where the data occupies a complex hyper-surface.

Figure 4.1a describes the usual Gaussian mixture model [1]. The components' centroids, $\boldsymbol{\mu} = (\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k)$, its covariances, $\boldsymbol{\Sigma}$, and the categorical selection distribution π, are outside the plate. It means we draw the centroids, variances, and the categorical distribution once, and use them for all the samples inside the plate. That constraints each category to be distributed around some fixed center which is pretty restrictive and perhaps too much so. By moving the random vectors inside the plate (figure 4.1b), we create a much more flexible distribution.

Another issue with the original mixture model, if we try to base a VAE model on it, is that it imposes a "bad categorical prior" on $\mathbf{y}$. We usually set $p(\mathbf{y}) \sim \text{Cat}(\pi)$ and $\pi = \frac{1}{k}$



(a) "vanilla" Gaussian mixture model with fixed centroids.

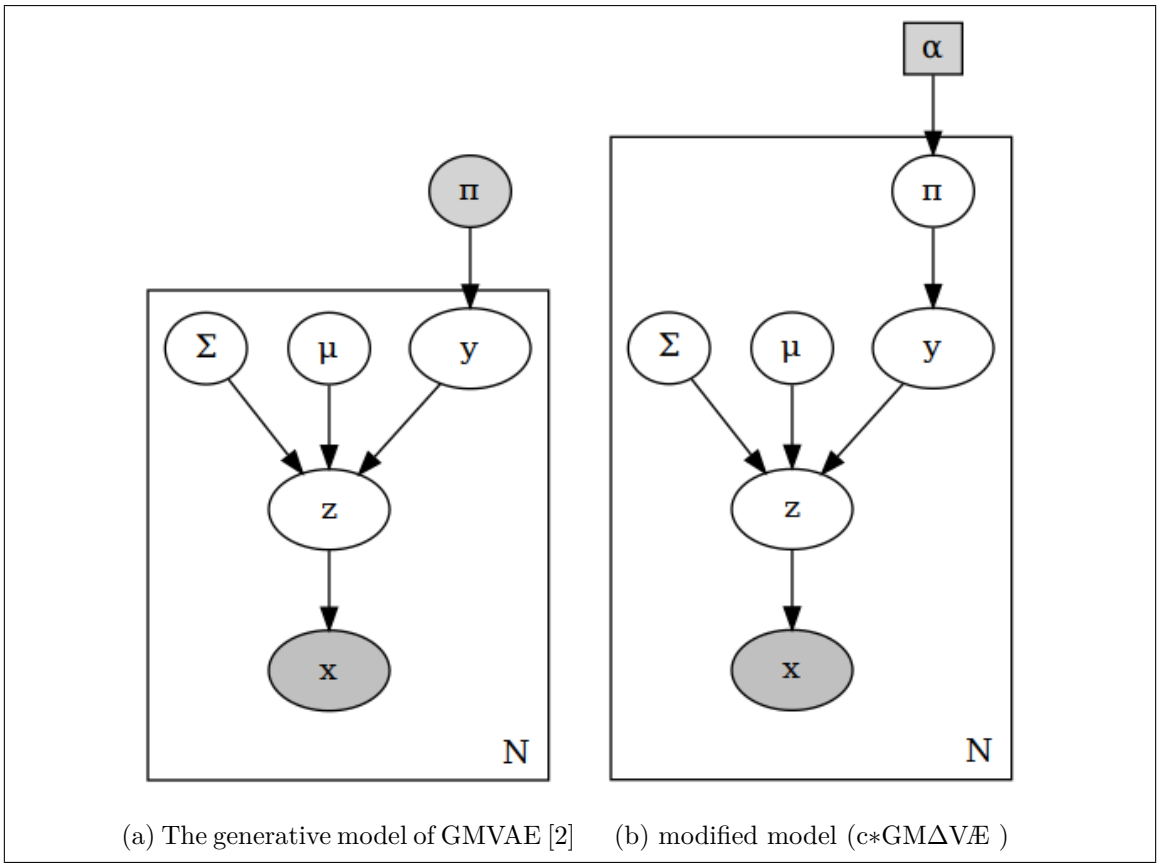


(b) a more complex mixture where the centroids are moved into the plate.

Figure 4.1

assuming the data is balanced. The problem is that we want the inference model (the encoder) $q(\mathbf{y}|\mathbf{x})$ to be a good classifier so we don't want to impose it to be close to uniform. If we move π inside and give it a symmetric Dirichlet prior, we allow more flexibility on π, so it can be be heavily biased towards one category, making it a good predictor. α is a symmetric Dirichlet hyper-parameter and "tweaking" with it has influence on the number of non-empty categories that the model finds during training.

Figure 4.2a shows the generative model of the GMVAE [2], where the mixtures components (μ, σ) are inside the plate but the selection distribution (π) is a fixed parameter which is outside the of the plate. The latent $\mathbf{z}$ space is a mixture distribution. As explained above, the $\mathbf{y}$-prior is "bad" because it will be uniform categorical, while we want to use the inference model $q(\mathbf{y}|\mathbf{z})$ to predict the category. In figure 4.2b π is moved inside the plate and becomes a variable. Its prior is a symmetric Dirichlet with hyper-parameter $\alpha \in \mathbb{R}$. In the actual models, there is another random vector $(\mathbf{w})$ with a diagonal standard normal Gaussian prior distribution, that is used for generating the mixture components Σ, μ by a neural network.



(a) The generative model of GMVAE [2] (b) modified model (c∗GMΔVÆ)

Figure 4.2: On the left we see the generative model used by GMVAE [2], where π is a fixed hyper parameter and resides outside of the plate. On the right: modification of the GMVAE model, where π moves inside the plate and becomes a variable with a symmetric Dirichlet prior.

4.2 c∗GMΔVÆ : GMVAE with Dirichlet prior

Our variation of GMVAE,which we name c∗GMΔVÆ , is based on figure 4.2b. The Δ stands for Dirichlet and the c∗ represents a regex and it stands for ”maybe conditional”. Think of it as if there are as many ’c’s in a model instance’s name as there are conditions in the dataset it was meant to process. As will be shown later, this model also has a conditional version but for now we formalise the simple case without conditions. The In addition to the latent variables $\mathbf{z}$ (mixture), and $\mathbf{y}$ (categorical), there are 2 more variables. The third latent variabel $\mathbf{w}$ has standard normal prior. With $\mathbf{w}$ the model generates the means and variances μ, σ for the mixture's components of $\mathbf{z}$, by a of neural network. A fourth variable $\mathbf{d}$ is the Dirichlet prior. The full generative model factors as (figure 4.3):

$$\begin{aligned} p(\mathbf{x},\mathbf{y},\mathbf{z},\mathbf{w},\mathbf{d}) &= p(\mathbf{x}|\mathbf{z})p(\mathbf{z}|\mathbf{w},\mathbf{y})p(\mathbf{y}|\mathbf{d})p(\mathbf{d})p(\mathbf{w}) \\ p(\mathbf{w}) &= \mathcal{N}(\mathbf{w}|\mathbf{0},\mathbf{1}) \\ p(\mathbf{d}) &= \mathrm{Dir}(\mathbf{d}|\alpha) \\ p(\mathbf{y}|\mathbf{d}) &= \mathrm{Cat}(\mathbf{y}|\mathbf{d}) \\ p(\mathbf{z}|\mathbf{w},\mathbf{y}) &= \mathcal{N}(\mathbf{z}|\mu(\mathbf{w})_{\mathbf{y}},\sigma(\mathbf{w})_{\mathbf{y}})) \\ p(\mathbf{x}|\mathbf{z}) &= \mathcal{N}(\mathbf{x}|\mu(\mathbf{z}),\sigma(\mathbf{z})) \end{aligned} \tag{4.1}$$

$\mu(\mathbf{w}), \sigma(\mathbf{w}), \mu(\mathbf{z}), \sigma(\mathbf{z})$ are neural networks of the decoder. $\mu(\mathbf{w})$, $(\sigma(\mathbf{w}))$ is not just one

mean (variance) but a set of k (=number of categories) means (variances), and we are selecting the component $\mu(\mathbf{w})_{\mathbf{y}}, \sigma(\mathbf{w})_{\mathbf{y}}$ indicated by $\mathbf{y}$, the categorical random variable.

The generative process can be described as follows:

1. draw from the prior $\mathbf{w} \sim p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \mathbf{1})$
2. draw a categorical distribution from the Dirichlet prior $\mathbf{d} \sim p(\mathbf{d}) = \mathrm{Dir}(\mathbf{d}|\alpha)$
3. draw a category $\mathbf{y} \sim p(\mathbf{y}|\mathbf{d}) = \mathrm{Cat}(\mathbf{y}|\mathbf{d})$
4. generate the k centroids and diagonal variances by the decoder's NN: $\boldsymbol{\mu}(\mathbf{w}), \boldsymbol{\sigma}(\mathbf{w})$.
5. draw from the mixture $\mathbf{z} \sim p(\mathbf{z}|\mathbf{w}, \mathbf{y}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}(\mathbf{w})_{\mathbf{y}}, \boldsymbol{\sigma}(\mathbf{w})_{\mathbf{y}})$
6. generate the means $\boldsymbol{\mu}(\mathbf{z})$ and diagonal variances $\boldsymbol{\sigma}(\mathbf{z})$ (or use fixed variance 1) by the decoder NN.
7. draw observation $\mathbf{x} \sim p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}(\mathbf{z}))$

There are several ways to factor the inference model and it is not immediately clear which one is better. Here is the factorization depicted in figure 4.3:

$$\begin{aligned} q(\mathbf{y}, \mathbf{z}, \mathbf{w}, \mathbf{d}|\mathbf{x}) &= q(\mathbf{z}|\mathbf{x})q(\mathbf{w}|\mathbf{x})q(\mathbf{y}|\mathbf{z})q(\mathbf{d}|\mathbf{z}) \\ q(\mathbf{z}|\mathbf{x}) &= \mathcal{N}(\mathbf{z}|\mu_z(\mathbf{x}), \sigma_z(\mathbf{x})) \\ q(\mathbf{w}|\mathbf{x}) &= \mathcal{N}(\mathbf{w}|\mu_w(\mathbf{x}), \sigma_w(\mathbf{x})) \\ q(\mathbf{y}|\mathbf{z}) &= \mathrm{Cat}(\mathbf{y}|f(\mathbf{z})) \\ q(\mathbf{d}|\mathbf{z}) &= \mathrm{Dir}(\mathbf{d}|g(\mathbf{z})) \end{aligned} \tag{4.2}$$

And $f(\mathbf{z}), g(\mathbf{z}), \mu_w(\mathbf{x}), \sigma_w(\mathbf{x}), \mu_z(\mathbf{x}), \sigma_z(\mathbf{x})$ are the neural networks of the encoder.

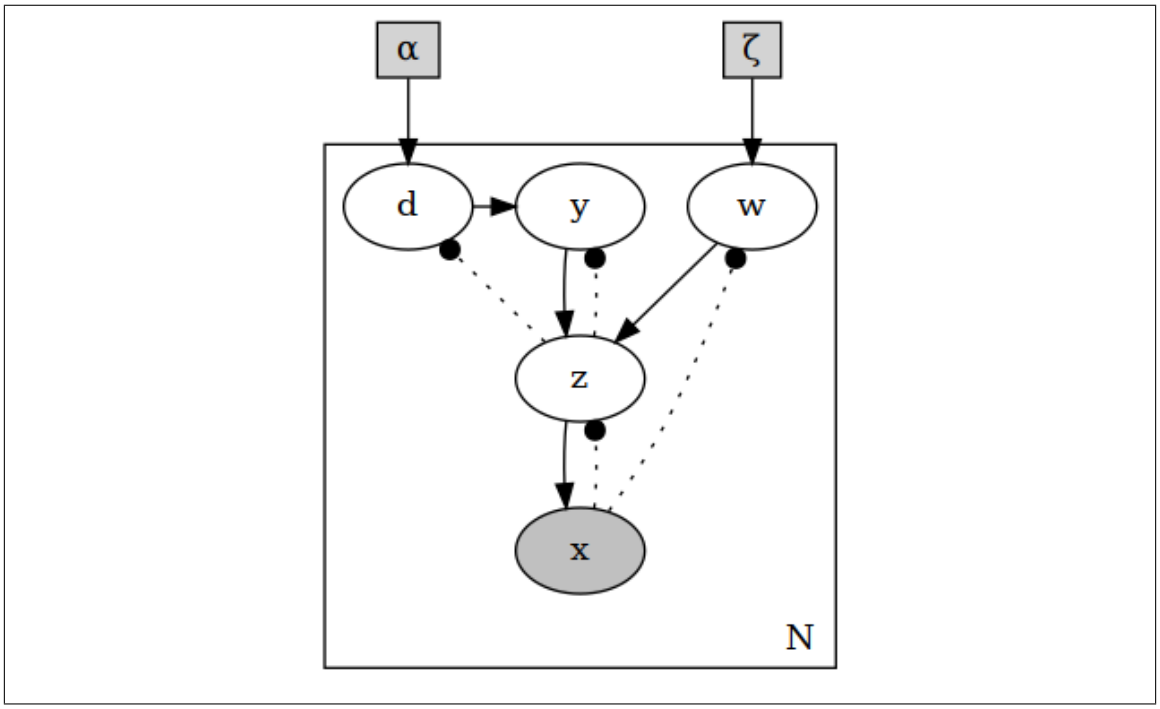


Figure 4.3: GMVAE with Dirichlet prior

Let's call the c∗GMΔVÆ model as presented in figure 4.3 our "canonical" version. The results we present here unless otherwise stated were obtained on the canonical model. As mentioned the choice of how to factor the inference (q) distribution is somewhat arbitrary and we have tested some other possibilities, for example by letting $\mathbf{x}$ play a direct role in the classifier $q(\mathbf{y}|\grave{)}$: $q(\mathbf{y}, \mathbf{z}, \mathbf{w}, \mathbf{d}, |\mathbf{x}) = q(\mathbf{z}|\mathbf{x})q(\mathbf{w}|\mathbf{x})q(\mathbf{y}|\mathbf{z}, \mathbf{x})q(\mathbf{d}|\mathbf{z})$

The canonical version is theoretically purer in the sense that since the mixture distribution is on the latent space $\mathbf{z}$ it should be possible to determine the class just from $\mathbf{z}$. The other version has the advantage that no information is lost on the encoding into $\mathbf{z}$ because the classifier observes both $\mathbf{z}$ and the original input. From tests we made we can't see a clear advantage to one factoring of q over others, from those we have tried.

Remark 4.1. In the original GMVAE paper [2], they don't use a neural network to define the classifier $q(\mathbf{y}|\mathbf{z})$. Actually their factorization is given as $q(\mathbf{y}|\mathbf{z}, \mathbf{w})$ but more importantly they rely they define $q(\mathbf{y}|\mathbf{z}, \mathbf{w}) \triangleq p(\mathbf{y}|\mathbf{z}, \mathbf{w})$ (they don't have Dirichlet prior d) because it is tractable:

$$p(\mathbf{y} = i|\mathbf{z}, \mathbf{w}) = \frac{p(\mathbf{y} = i)p(\mathbf{z}|\mathbf{w}, \mathbf{y} = i)}{\sum_{j=1}^{K} p(\mathbf{y} = j)p(\mathbf{z}|\mathbf{w}, \mathbf{y} = j)}$$

There is at least one implementation [16] of GMVAE which doesn't use this tractability in q.

In c∗GMΔVÆ with the addition of a Dirichlet prior, we have $q(\mathbf{y}|)$ and it isn't possible, or at least not immediately obvious, how to use $p(\mathbf{y}|\mathbf{w}, \mathbf{z})$ in the q factorization. Moreover p and q are two separate distributions, and even though $p(\mathbf{y}|\mathbf{w}, \mathbf{z})$ is tractable when $\mathbf{y}$ happens to have discrete distribution, it doesn't mean we must impose it on q.

4.3 Computing the loss function of c∗GMΔVÆ

The loss function remains the -ELBO and we can break it into different terms:

$$\begin{aligned}
\mathcal{L}(p, q, \mathbf{x}) &= \int -\log \frac{p(\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{w}, \mathbf{d})}{q(\mathbf{z}, \mathbf{y}, \mathbf{w}, \mathbf{d}|\mathbf{x})} dq(\mathbf{z}, \mathbf{y}, \mathbf{w}, \mathbf{d}|\mathbf{x}) && (4.3)\\
&= \int -\log \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z}|\mathbf{w}, \mathbf{y})p(\mathbf{y}|\mathbf{d})p(\mathbf{w})p(\mathbf{d})}{q(\mathbf{z}|\mathbf{x})q(\mathbf{w}|\mathbf{x})q(\mathbf{y}|\mathbf{z})q(\mathbf{d}|\mathbf{z})} dq && (4.4)\\
&= \int -\log p(\mathbf{x}|\mathbf{z}) dq && (4.5)\\
&+ \int \log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{w}, \mathbf{y})} dq && (4.6)\\
&+ \int \log \frac{q(\mathbf{w}|\mathbf{x})}{p(\mathbf{w})} dq && (4.7)\\
&+ \int \log \frac{q(\mathbf{y}|\mathbf{z})}{p(\mathbf{y}|\mathbf{d})} dq && (4.8)\\
&+ \int \log \frac{q(\mathbf{d}|\mathbf{z})}{p(\mathbf{d})} dq && (4.9)
\end{aligned}$$

As we see there are five terms in the reconstruction loss function. 4.14 is called the reconstruction error. The rest of the terms get the repetitive, mundane names: 4.15 is called the z-error or z-kl-term, 4.16 is the w-error or w-kl-term, 4.18 is the y-error or y-kl-term, and 4.9 is called the d-error or d-kl-term.

To actually compute the loss function Monte Carlo integration is used, whereby *one* sample is taken for each observation. We need to sample $\mathbf{z}$, $\mathbf{w}$ and $\mathbf{d}$. As for $\mathbf{y}$ we don't sample but rather integrate over the categorical probabilities.

1. sample $\mathbf{z} \sim q(\mathbf{z}|\mathbf{x})$ using the reparameterization trick
2. sample $\mathbf{w} \sim q(\mathbf{w}|\mathbf{x})$ using the reparameterization trick
3. sample $\mathbf{d} \sim q(\mathbf{d}|\mathbf{z})$ using the reparameterization trick
4. Reconstruction error estimation:
$$-\log p(\mathbf{x}|\mathbf{z}) = -\log \mathcal{N}(\mathbf{x}|\mu(\mathbf{z}), \sigma(\mathbf{z}))$$
5. $\mathbf{z}$-error: calculate $KL(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{w},\mathbf{y}))$ analytically using equation 3.13 for every mixture component of $p(\mathbf{z}|\mathbf{w},\mathbf{y})$, and then take the weighted average with respect to $q(\mathbf{y}|\mathbf{z})$:
$$\sum_{j=1}^{k}[q(\mathbf{y}=j|\mathbf{z})KL(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{w},\mathbf{y}=j))]$$
.
6. $\mathbf{w}$-error: calculate $KL(q(\mathbf{w}|\mathbf{x})\|p(\mathbf{w}))$ analytically the KL-divergence between two diagonal Gaussians (equation 3.13).
7. $\mathbf{y}$-error: calculate $KL(q(\mathbf{y}|\mathbf{z})\|p(\mathbf{y}|\mathbf{d}))$ analytically (kl-div of two categorical distributions,):
$$KL(q(\mathbf{y}|\mathbf{z})\|p(\mathbf{y}|\mathbf{d})) = \sum_{j=1}^{k}[q(\mathbf{y}=j|\mathbf{z})\log\frac{q(\mathbf{y}=j|\mathbf{z})}{p(\mathbf{y}=j|\mathbf{d})}]$$
8. $\mathbf{d}$-error: calculate $KL(q(\mathbf{d}|\mathbf{z})\|p(\mathbf{d}))$ analytically (KL-divergence of two Dirichlet distributions also has a closed formed formula and it's builtin function in Pytorch).

4.4 c∗GMΔVÆ in the supervised case

c∗GMΔVÆ can also be used for supervised training. While the architecture of the neural network remains the same, the probabilistic interpretation and consequently the calculation of the loss function is different.

Figure 4.4 describes the generative and the inference model in the supervised case. Since in this case $\mathbf{y}$ is observed, we regard $\mathbf{z}$ in the generative model as only dependent on $\mathbf{w}$. Internally we keep the same architecture but during supervised training we always select just the given $\mathbf{y}$ component and so we can stop regarding $\mathbf{z}$ as depending on $\mathbf{y}$. $p(\mathbf{z}|\mathbf{w},\mathbf{y}) = p(\mathbf{z}|\mathbf{w}) = \mathcal{N}(\mathbf{z}|\mu(\mathbf{w})_y, \sigma(\mathbf{w})_y)$. In the figure we show the dashed arrow from $\mathbf{y}$ to $\mathbf{w}$ but if it were a real dependency we would get an "illegal" cycle in the p-graph. However the $\mathbf{y}$ which we used to select the $\mu(\mathbf{w}), \sigma(\mathbf{w})$ mixture components are not drawn

from $p(\mathbf{y}|\mathbf{z})$, rather we use the observation $\mathbf{y}$ (=the category label) itself which is given to us in the supervised case. Thus there is no actual cycle in the dependency graph but the effect of $\mathbf{y}$ remains in the implementation of the network architecture.

Moreover in the supervised case the random variable $\mathbf{y}$ becomes dependent on both $\mathbf{d}$ and $\mathbf{z}$. We make an assumption that $\mathbf{y}|\mathbf{d}$ and $\mathbf{y}|\mathbf{z}$ are independent, so $p(\mathbf{y}|\mathbf{d},\mathbf{z}) = p(\mathbf{y}|\mathbf{d})p(\mathbf{y}|\mathbf{z})$, thus enabling us to retain the same network architecture. $p(\mathbf{y}|\mathbf{d})$ is determined by the same decoder component as in the unsupervised case but $p(\mathbf{y}|\mathbf{z})$ equals $q(\mathbf{y}|\mathbf{z})$ of the encoder in the unsupervised case. So while again we keep the same architecture we will compute the loss function differently. As for the inference model other than losing the $q(\mathbf{y}|\mathbf{z})$ component the rest remains the same. The full factorization in the supervised case becomes:

$$
\begin{aligned}
p(\mathbf{x},\mathbf{y},\mathbf{z},\mathbf{w},\mathbf{d}) &= p(\mathbf{x}|\mathbf{z})p(\mathbf{y}|\mathbf{d})p(\mathbf{y}|\mathbf{z})p(\mathbf{z}|\mathbf{w})p(\mathbf{d})p(\mathbf{w}) \\
q(\mathbf{z},\mathbf{w},\mathbf{d}|\mathbf{x},\mathbf{y}) &= q(\mathbf{z}|\mathbf{x})q(\mathbf{w}|\mathbf{x})q(\mathbf{d}|\mathbf{z}) \\
p(\mathbf{x}|\mathbf{z}) &= \mathcal{N}(\mathbf{x}|\mu(\mathbf{z}),\sigma(\mathbf{z})) \\
p(\mathbf{y}|\mathbf{z}) &= \mathrm{Cat}(\mathbf{y}|f(\mathbf{z})) \\
p(\mathbf{y}|\mathbf{d}) &= \mathrm{Cat}(\mathbf{y}|\mathbf{d}) \\
p(\mathbf{w}) &= \mathcal{N}(\mathbf{w}|\mathbf{0},\mathbf{1}) \\
p(\mathbf{d}) &= \mathrm{Dir}(\mathbf{d}|\alpha) \\
p(\mathbf{z}|\mathbf{w}) &= \mathcal{N}(\mathbf{z}|\mu(\mathbf{w})_{\mathbf{y}},\sigma(\mathbf{w})_{\mathbf{y}})) \\
q(\mathbf{z}|\mathbf{x}) &= \mathcal{N}(\mathbf{z}|\mu_z(\mathbf{x}),\sigma_z(\mathbf{x})) \\
q(\mathbf{w}|\mathbf{x}) &= \mathcal{N}(\mathbf{w}|\mu_w(\mathbf{x}),\sigma_w(\mathbf{x})) \\
q(\mathbf{d}|\mathbf{z}) &= \mathrm{Dir}(\mathbf{d}|g(\mathbf{z}))
\end{aligned}
\tag{4.10}
$$

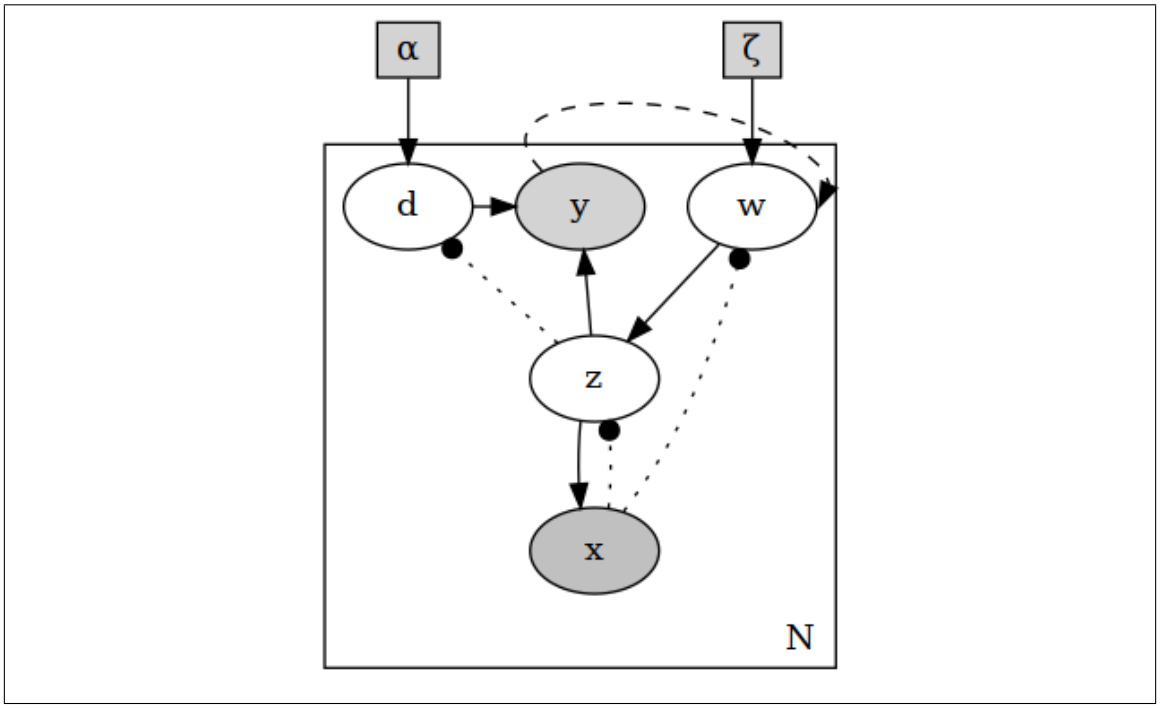


Figure 4.4: c∗GMΔVÆ : the supervised case, where $\mathbf{y}$ is an observed variable.

The loss function in the supervised case is:

$$\mathcal{L}(p,q,\mathbf{x}) = \int -\log \frac{p(\mathbf{x},\mathbf{y},\mathbf{z},\mathbf{w},\mathbf{d})}{q(\mathbf{z},\mathbf{w},\mathbf{d}|\mathbf{x},\mathbf{y})} dq(\mathbf{z},\mathbf{y},\mathbf{w},\mathbf{d}|\mathbf{x}) \tag{4.11}$$

$$= \int -\log \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z}|\mathbf{w})p(\mathbf{y}|\mathbf{d})p(\mathbf{y}|\mathbf{z})p(\mathbf{w})p(\mathbf{d})}{q(\mathbf{z}|\mathbf{x})q(\mathbf{w}|\mathbf{x})q(\mathbf{d}|\mathbf{z})} dq \tag{4.12}$$

$$= \int -\log p(\mathbf{x}|\mathbf{z}) dq \tag{4.13}$$

$$+ \int \log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{w})} dq \tag{4.14}$$

$$+ \int \log \frac{q(\mathbf{w}|\mathbf{x})}{p(\mathbf{w})} dq \tag{4.15}$$

$$+ \int \log p(\mathbf{y}|\mathbf{z}) dq \tag{4.16}$$

$$+ \int \log p(\mathbf{y}|\mathbf{d}) dq \tag{4.17}$$

$$+ \int \log \frac{q(\mathbf{d}|\mathbf{z})}{p(\mathbf{d})} dq \tag{4.18}$$

The reconstruction error, the w-error and the d-error are calculated exactly like in the unsupervised loss. The z-term is also calculated in exactly the same procedure as in unsupervised case except that we only need to take the component indicated by the obseved $\mathbf{y}$ in the weighted sum. The two y-terms are the log probability of the observed category given the sampled $\mathbf{z}$ and $\mathbf{d}$.

4.5 Combining c∗GMΔVÆ with another clustering method

Because c∗GMΔVÆ performs particularly well in (semi)supervised it can learn a given cluster/class assignment of a dataset by. But when might it be useful to train c∗GMΔVÆ if we already have a different clustering algorithm or any other unsupervised classifier that works well? Suppose that we have a ”nice” training set where a given clustering algorithm performs well on. We can supervised train the model on the nice dataset using the cluster assignment as labels. At this point the model should learn to represent the data clusters nearly perfect. It is often the case that the c∗GMΔVÆ trained in this manner will outperform a ”naive” model which was unsupervised trained.

We can than run unsupervised training on the dataset, if we are lucky the accuracy might improve. So that's one use case but a minor one.

The more important usage is that we can now classify with c∗GMΔVÆ other datasets of the same kind. We should compare the performance of the trained model on the testing set with the performance of the clustering algorithm *on the testing set* which per assumption was not that good. If we are lucky the model might outperform the clustering algorithm on the not–so–nice testing data.

The combination of c∗GMΔVÆ with Louvain [21] clustering was tested on several datasets (results shown in subsequent chapters).

4.6 The conditional version of c∗GM∆VÆ

We have seen two types of models that deal with data that is both numerical and categorical, namely the conditional VAE (CVAE), and the c∗GM∆VÆ . As we hinted before, there is a conditional variation of c∗GM∆VÆ which is meant to be used with data that contains both "condition" and "class" information. While both are types of categorical data, "condition" is used to described information that is always visible in the data. This can be for example batch information. It is always known from which "batch" a cell was taken. "Class" information might not always be known, and it is more an inherent property and not "technical" difference. For example cell types are best represented as classes. It is technically very easy to implement a conditional version from a given VAE model. Basically it just requires concatenating the condition c to whatever variable we want to condition (which are all the variables in this case).

But why do we need it? Suppose that we have again numerical/categorical type of data $(\mathbf{x}, \mathbf{y})$ which comes in two or more types of "flavors" c, so the full data type is $(\mathbf{x}, \mathbf{y}, c)$. For example $\mathbf{x}$ can be an image of on of k different faces, $\mathbf{y}$ is the identity of the person, c is an indicator of whether the person smiles or not. Also suppose that $\mathbf{y}$ might not be provided at all or just for a subset of the data but c is always provided for us.

The data we purposed this model for is the Kang scRNASeq dataset [7] that comes from two types of conditions (control and stimulated), The condition represents the states before and after exposure to some treatment (INF-β). $\mathbf{x}$ is normalized gene expression levels, $\mathbf{y}$ is the cell type we want to infer but may not know the ground truth, and c indicates from which group the measurement was taken (control/stimulate) and this information is visible to use because we know for each sample from which group it was taken.

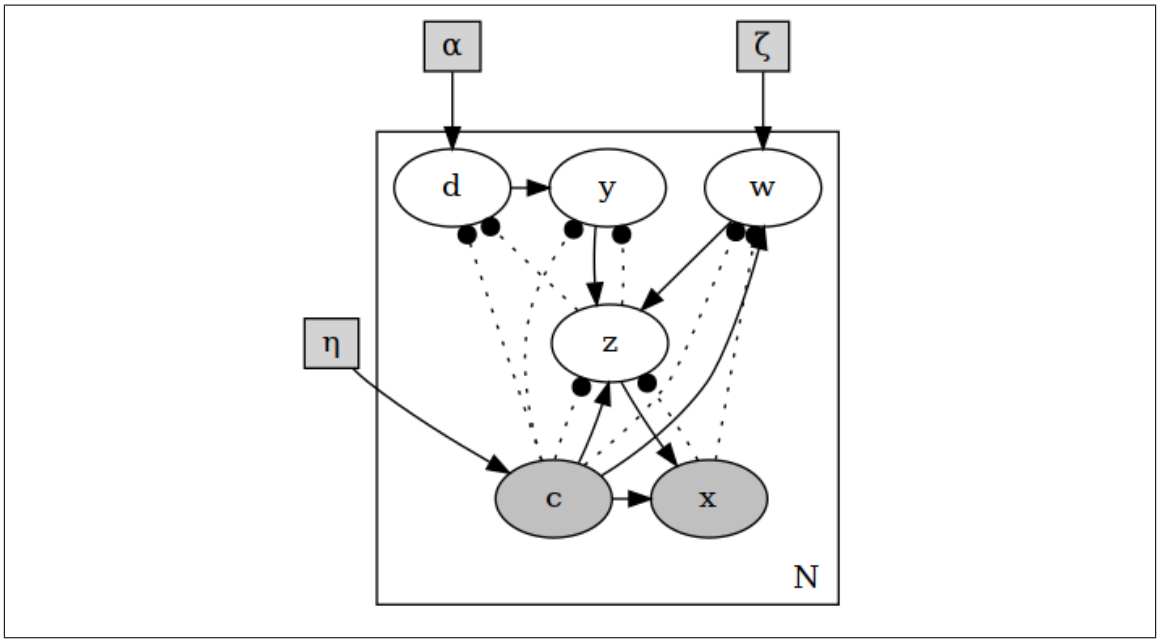


Figure 4.5: c∗GM∆VÆ (cond.), unsupervised case. Sorry for the arrow clutter

The factorization in the unsupervised case is:

$$\begin{aligned} p(\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{w}, \mathbf{d}|c) &= p(\mathbf{x}|\mathbf{z}, c)p(\mathbf{z}|\mathbf{w}, \mathbf{y}, c)p(\mathbf{y}|\mathbf{d}, c)p(\mathbf{d}|c)p(\mathbf{w}|c) \\ q(\mathbf{z}, \mathbf{w}, \mathbf{d}|\mathbf{x}, \mathbf{y}, c) &= q(\mathbf{z}|\mathbf{x}, c)q(\mathbf{w}|\mathbf{x}, c)q(\mathbf{y}|\mathbf{z}, c)q(\mathbf{d}|\mathbf{z}, c) \end{aligned} \tag{4.19}$$

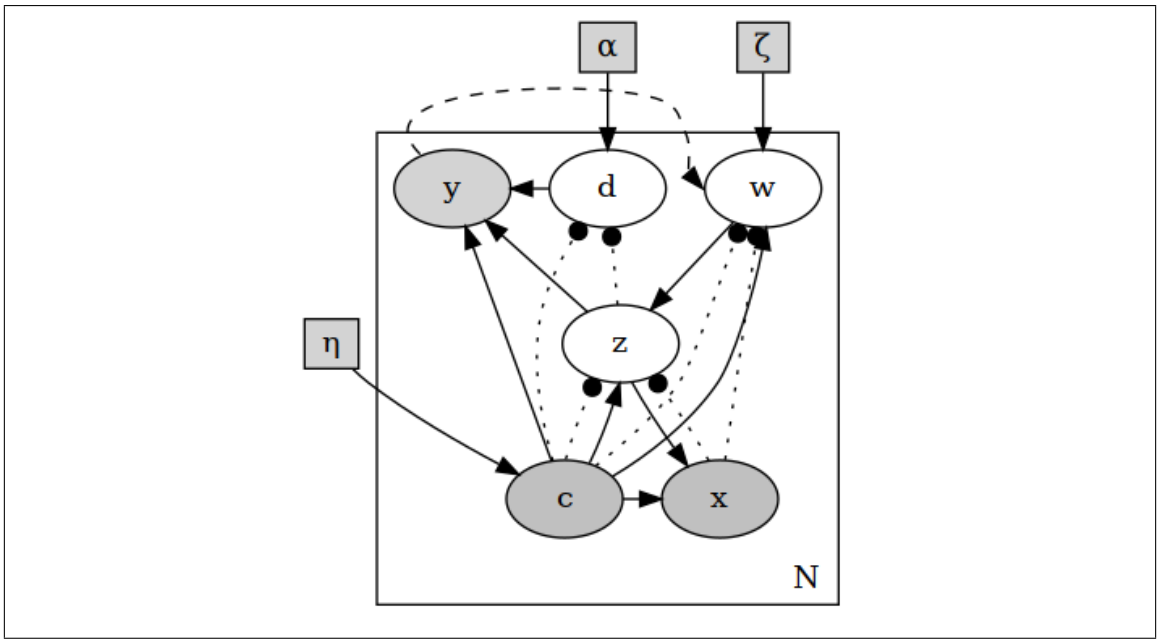


Figure 4.6: c∗GMΔVÆ (cond.), supervised case

And in the supervised case:

$$\begin{aligned} p(\mathbf{x},\mathbf{y},\mathbf{z},\mathbf{w},\mathbf{d}|c) &= p(\mathbf{x}|\mathbf{z},c)p(\mathbf{y}|\mathbf{d},c)p(\mathbf{y}|\mathbf{z},c)p(\mathbf{z}|\mathbf{w},c)p(\mathbf{d}|c)p(\mathbf{w}|c) \\ q(\mathbf{z},\mathbf{w},\mathbf{d}|\mathbf{x},\mathbf{y},c) &= q(\mathbf{z}|\mathbf{x},c)q(\mathbf{w}|\mathbf{x},c)q(\mathbf{d}|\mathbf{z},c) \end{aligned} \tag{4.20}$$

In this model we in particularly want to keep the dependency of $\mathbf{x}$ on c, hence $p(\mathbf{x}|\mathbf{z},c)$. This allows us to "flip" condition of the reconstruction, which we can interpret as converting for example control cells to stimulated cells, as we'll see in a later chapter.

Chapter 5

Tests on MNIST

The first dataset we tested the model on was MNIST [24] which is easy to machine learn yet it is also the most tested and compared dataset. This is a collection of 28×28 pixel gray–scale images of hand written digits. The images are labeled with the ground truth so it is an $(\mathbf{X}, \mathbf{Y})$ kind of data, with $\mathbf{X}$ being numerical and $\mathbf{Y}$ categorical.

MNIST images are close to being binary black and wight (either close to 0 or close to 1), so it is better to model them with Bernoulli distribution than with Gaussian, with the intensity of each pixel is regarded as a Bernoulli probability. Unless otherwise stated, for MNIST we set $p(\mathbf{x}|\mathbf{z})$ to be Bernoulli distribution.

5.1 semi-supervised learning

In the first test we partition the training set into labeled (5% of the total) and unlabeled subset (the other 95%). As we have some labeled images, we set the number–of–categories hyperparameter on 10 and the goal here is to be able to predict the true categorical partitioning of the testing set.

There are several qualitative aspects that we are particularly interested in this experiment:

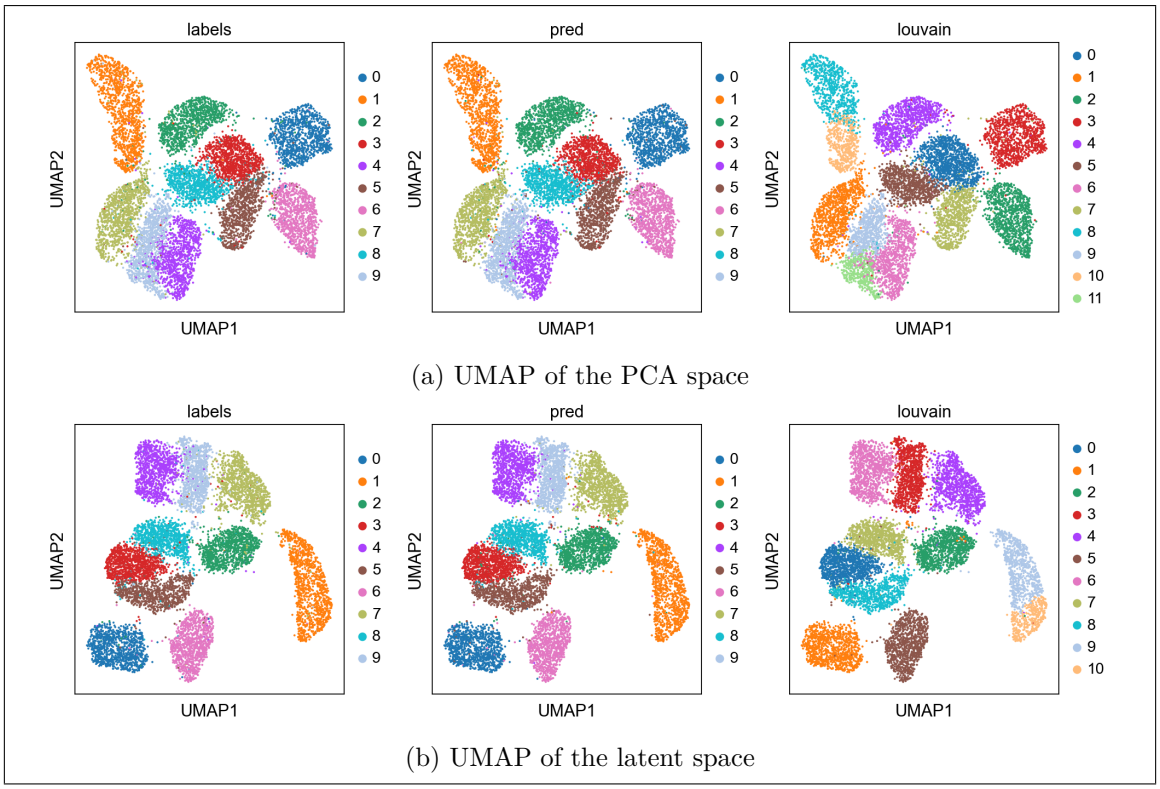
- The overall prediction accuracy of the model
- Comparing the models predicted clusters to Louvain clusters

  in the PCA space

  in the latent space

The overall accuracy of the model is 0.955 and as the figures 5.1 the prediction basically identifies the real classes. When performed on the PCA space, Louvain clustering with default settings identifies 12 clusters . The ”0” and ”9” classes each are split between two Louvain clusters. The Louvain clusters themselves appear to be homogeneous (each cluster contain practically only samples from one ground-truth class).

It is interesting to compare the PCA results to the same procedure done on the latent space. The UMAP appears a little bit better (the ground truth classes have more



(a) UMAP of the PCA space

(b) UMAP of the latent space

Figure 5.1: Comparison of the predictive quality of the model vs. Louvain clustering on the PCA space (top) and on the latent space (bottom).

separation from each other). This time there are 11 Louvain clusters and only the "9" category is split between two Louvain clusters. This is despite the latent z space being only 16-dimensional, while the PCA is done on the 50 most significant components.

Figure 5.2 shows the decoder's reconstruction of random samples from the mixture distribution which was imposed on the latent space. As we can see each component generates exactly one of the digit classes. Sampling from the latent mixture distribution (z-space) is done as follows: Firs we take random samples from the w-space, on which we imposed a standard normal diagonal distribution. Each sample corresponds to one row in the image. Then each w-sample is projected by the decoder into 10 (number of defined classes) mixture component. Each of these component if then further projected by the decoder network into the sample space (x).

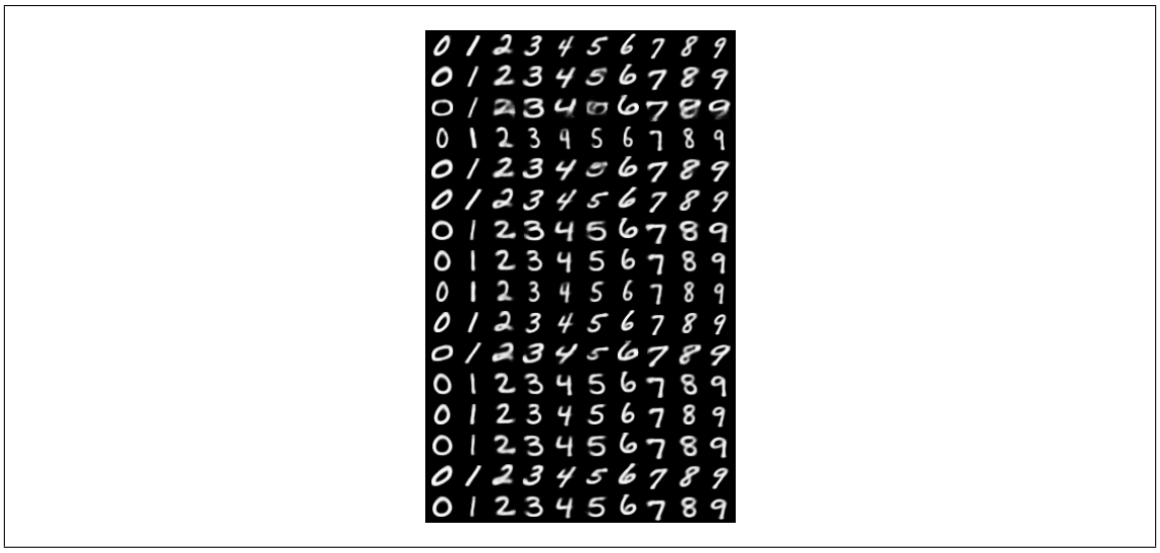


Figure 5.2: Images generated by random sampling from the latent mixture distribution. As we can see it fits perfectly with the right ground truth classes.

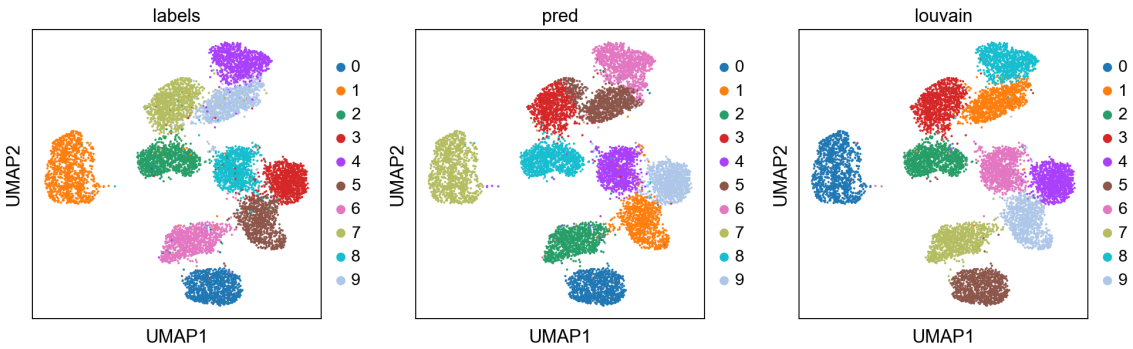
5.2 Unsupervised learning

5.2.1 10 Clusters (exact clustering)

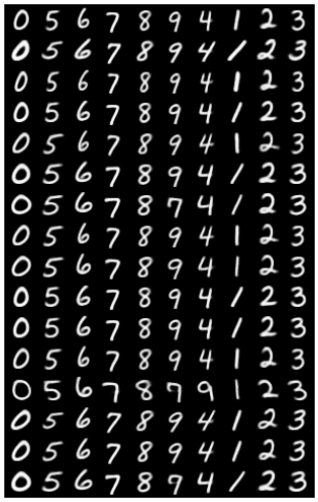
Getting the model to learn exactly the 10 ground truth classes is a daunting task. It requires a lot of tuning and micromanaging of the training procedure. In my experiments, the model first needs to be trained with standard learning rate ($1e-3$) to the point where it can adequately reconstruct digits and uses all 10 classes. Then the learning rate should be reduced to the range $5e-2$ — $5e-3$. This makes reconstruction less exact and allows the model to "jump" class predictions during training. If the learning rate is too big eventually the model might completely drift away and become inaccurate and unstable and if it is not sufficiently large the model remains stuck on a suboptimal classification. After this course training finds the correct classes, it needs to be further trained to improve accuracy and reconstruction with decreasing learning rates.

In my experiments, an accuracy of about 0.9—0.95 can be achieved however due the micromanagement involved it is hard to make the training fully automated. Moreover it may be completely unrealistic to expect the model to perform that well in unsupervised exact clustering tasks with harder datasets.

Figures 5.3 shows the results of unsupervised, exact clustering of MNIST by the c∗GMΔVÆ with 8 dimensional latent z-space and w-space. As both the UMAP and the generated samples show, there is still a little bit inaccuracies left, mainly with "7","4", and "9" digits. Despite this, the actual latent space is very nicely clustered and louvain clustering over this space retrieves the true classification of the the data set, even to higher accuracy than the model prediction. Note that in unsupervised learning the predicted classes are naturally untagged with the ground truth class tag, the point is that it should find essentially the true partitioning of the data. The tag assignment can be done later. for example each predicted cluster is tagged with the class with which it has the most overlap.



(a) UMAP of the latent space



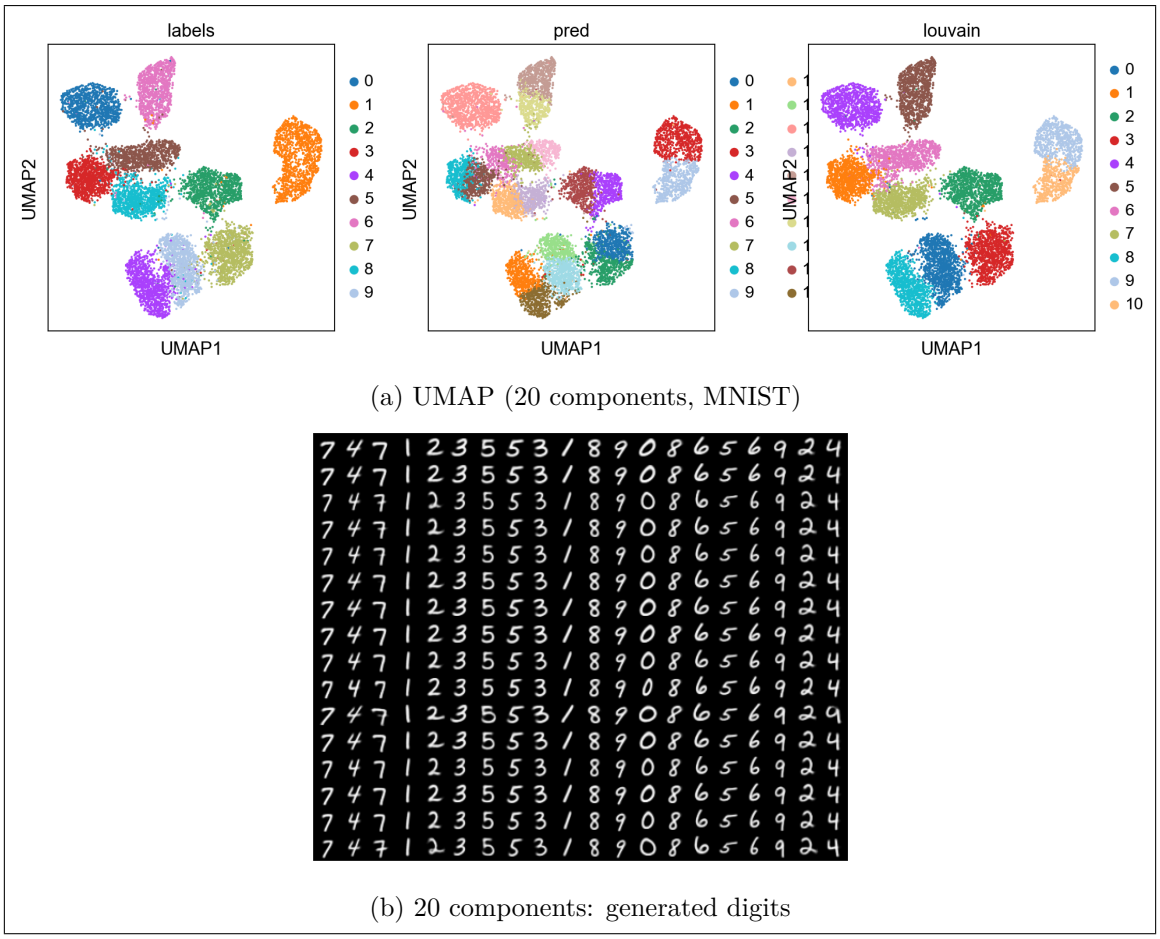
(b) Randomly generated samples

Figure 5.3: c∗GMΔVÆ : unsupervised learning of MNIST. This particular model achieved 0.93 accyracy which probably could be a little bit further improved with fine-tune training. The Louvain clustering of the latent space is even better than the prediction.

5.2.2 With overclustering

Overclustering means defining a mixture distribution with more components then ground truth classes. This helps the model to create homogenous classes, where each component is nearly a subset of a single class. Moreover In the case of MNIST each digit has actually 2 or more basic appearance (e.g. strait vs. italic version, closed vs. open top ”4” figure etc.). One can argue that there are more ”natural” classes in this dataset than the 10 digits. We ran unsupervised training of a c∗GMΔVÆ with 20 components.

Figures 5.4 show the results of the 20–component c∗GMΔVÆ . ”0” constitutes its own cluster, while ”5” is split between 3 clusters. The rest of the classes are each split between two clusters. Each cluster has its own digit-style (e.g. a strait ”7” in one cluster, and more cursive version in the other cluster). Unfortunately there is some misclassifications between ”4” and ”9”. Overall accuracy is about 0.946, probably with some room for improvement with additional training rounds. Louvain clustering of the latent space is able to almost perfectly capture the true classes, with the ”1” class split between two Louvain clusters and the other classes correspond each to a single Louvain cluster.



(a) UMAP (20 components, MNIST)

(b) 20 components: generated digits

Figure 5.4: c∗GMΔVÆ with 20 components, unsupervised learning of MNIST.

Training an overclustering model provides much more consistent results without training micromanagement. It is probably advisable to set up a model with more mixture components than the ground truth classes when trying unsupervised classification of more complex datasets.

5.3 Combination with Louvain clustering

Louvain clustering achieves on its own an accuracy of 0.967 on the *training set* which is larger (60k images) and 0.925 on the smaller (10k) testing set.

Then supervised training of c∗GMΔVÆ using Louvain clusters as the labels was performed. After this step, the model was 0.966 accurate *on the testing set*. After further unsupervised training of the model the accuracy marginally improved to 0.968.

MNIST is an easy data set but still the improvement over Louvain from 0.92 to 0.968 is noticeable. There was also an improvement over the unsupervised model and it basically solved the misclassification issue between 9,7 and 4 which is the main problem we observes with unsupervised learning as we see in figure 5.5.

Figure 5.5: c∗GMΔVÆ with 12 components guided by Louvain clustering: randomly generated digits

5.4 Comparing with other methods

The results obtained in the preceding tests are comparable with the original GMVAE method [2]. Figure 5.6 is taken from the original paper of Dilokthanakul et. al.

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

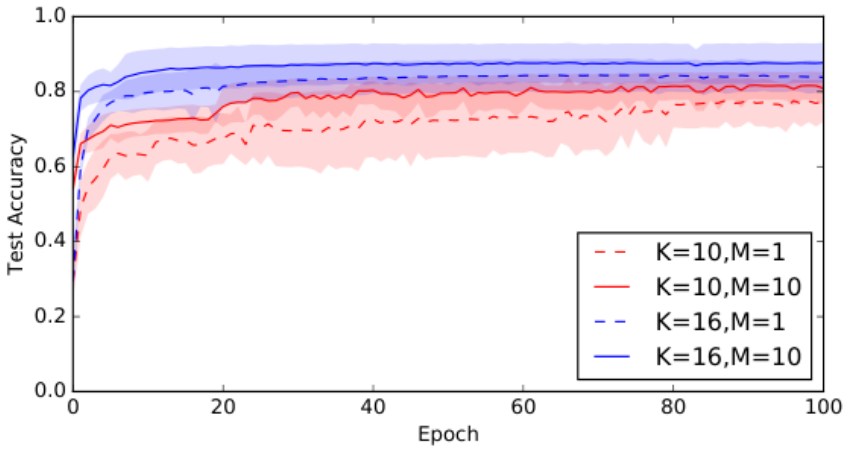


Figure 4: **Clustering Accuracy with different numbers of clusters (K) and Monte Carlo samples (M)** : After only few epochs, the GMVAE converges to a solution. Increasing the number of clusters improves the quality of the solution considerably.

Figure 5.6: Comparison of results by GMVAE and other methods as reported by [2]

Chapter 6

More challenging tests with fashion-MNIST

Fashion-MNIST (FMNIST) [3] is a drop-in replacement for MNIST which is a lot more challenging to machine–learn. Instead of digits we are dealing here with greyscale images of clothing and accessory items of 10 classes: *T-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, boot.* Although each image is tagged with a single category, it is not always obvious even for a human observer whether a certain image is actually a coat or a shirt, for example. There might be a little bit of this type of unclaritty in MNIST too, but it is much more pronounced in the fashion-MNIST dataset.

6.1 Unsupervised learning

c∗GMΔVÆ achieved an overall 0.75 accuracy after *a lot* of training epochs. It is easier to achieve about 0.67 accuracy after a quick training (about 10 epochs). Choice of hyperparameters also have significant effect. For example choosing a high dimensional latent space or deepening the networks tend to make the model perform to not use all the components of the mixture.

Judging from the randomly generated images in figure 6.1, the model learns to generate what looks like a distinct classes in each of the mixture components. However as far as prediction accuracy, there is a major confusion between ”shirt”, ”coat”, and ”pullover” classes, which is apparent from the UMAP in figure 6.2.

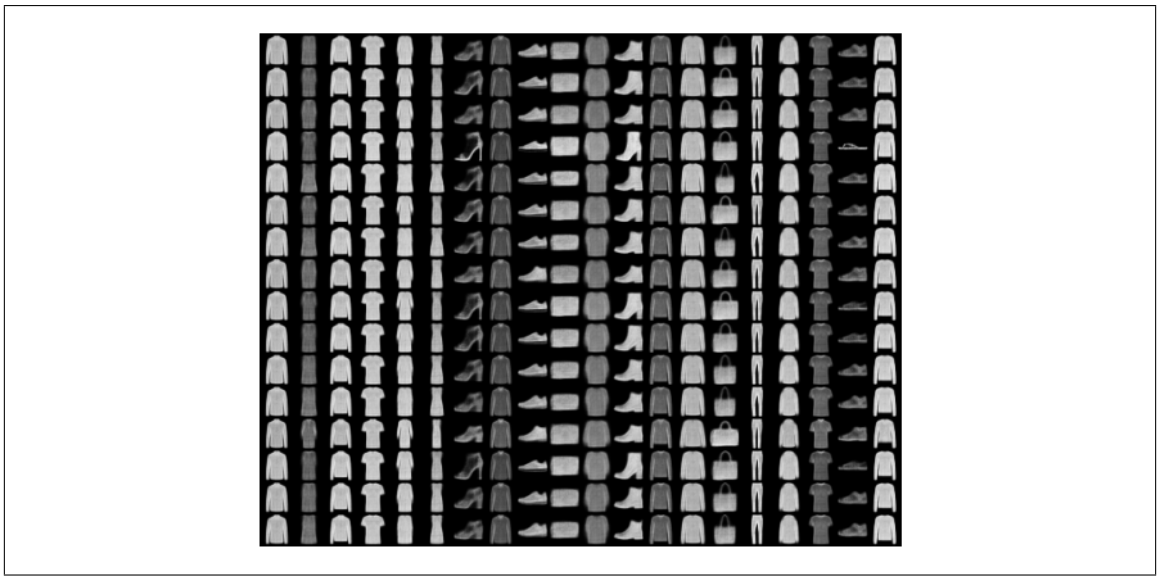


Figure 6.1: 20 components: generated digits

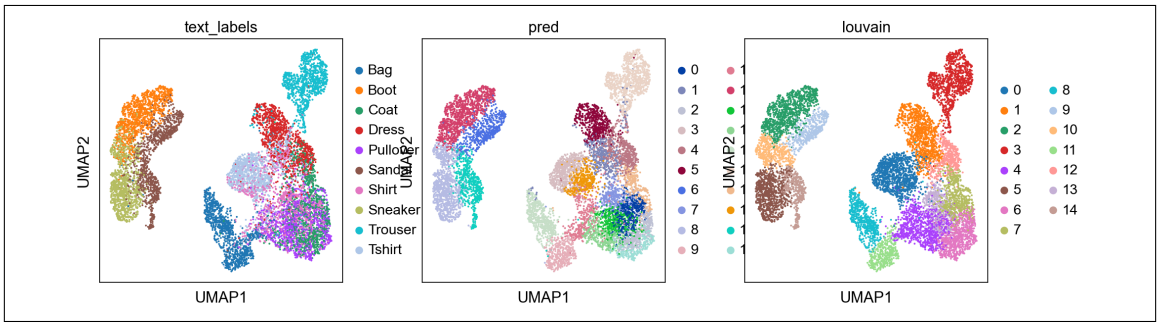


Figure 6.2: UMAP (20 components, MNIST)

6.2 Semisupervised learning

Our c∗GMΔVÆ achieved accuracy of 0.85 on semisupervised learning with 0.1 — 0.9 split between known and hidden labels. Considering that this accuracy was semisupervised and that the network layer architecture which was used was fully connected with just 2 hidden layers I think this accuracy is not bad.

Figure 6.3 shows randomly generated samples by the model. The 10 components match the ground truth classes fairly well.

Figure 6.4 shows UMAP of the laten space. The top-wear (shirt, coat, pullover, T-shirt) are still pretty mixed and that's where most of the misclassifications occur.

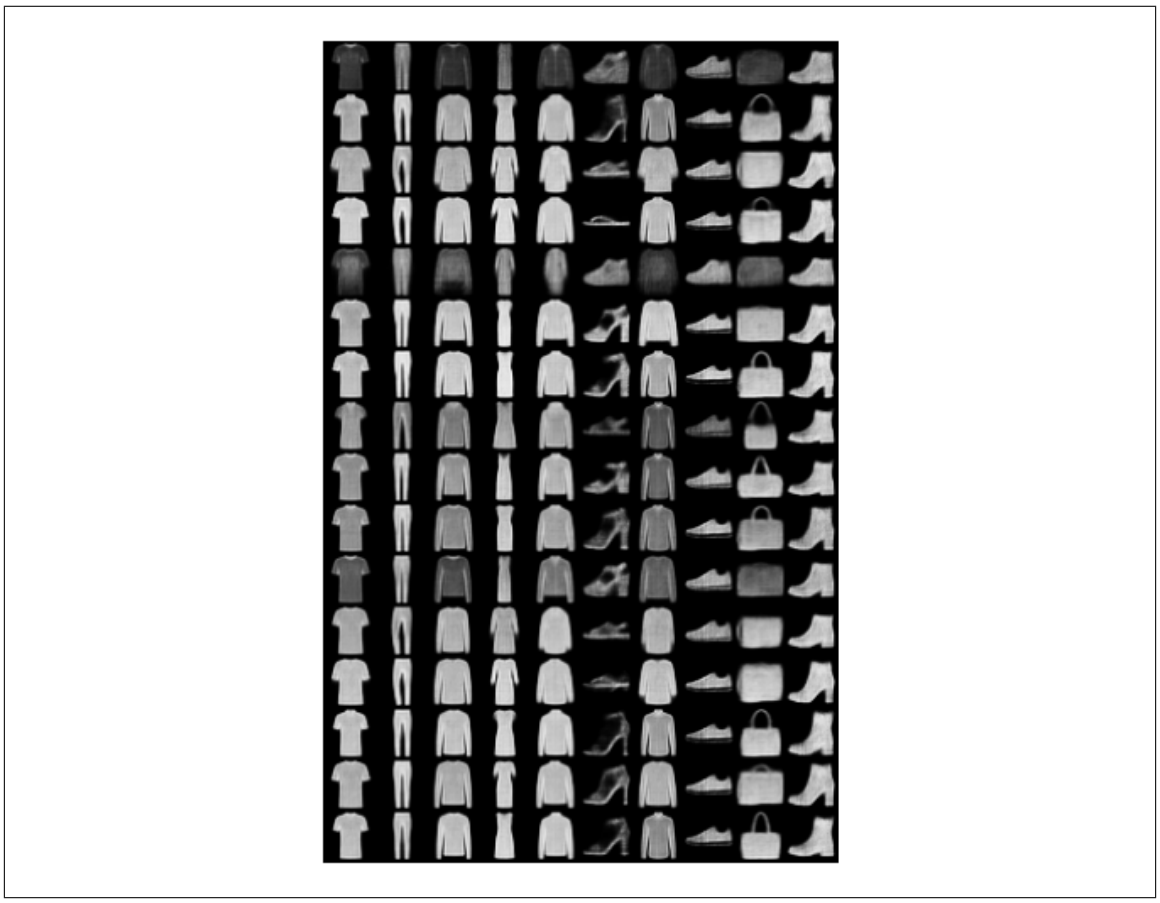


Figure 6.3: FMNIST, semisupervised. Randomly generated samples.

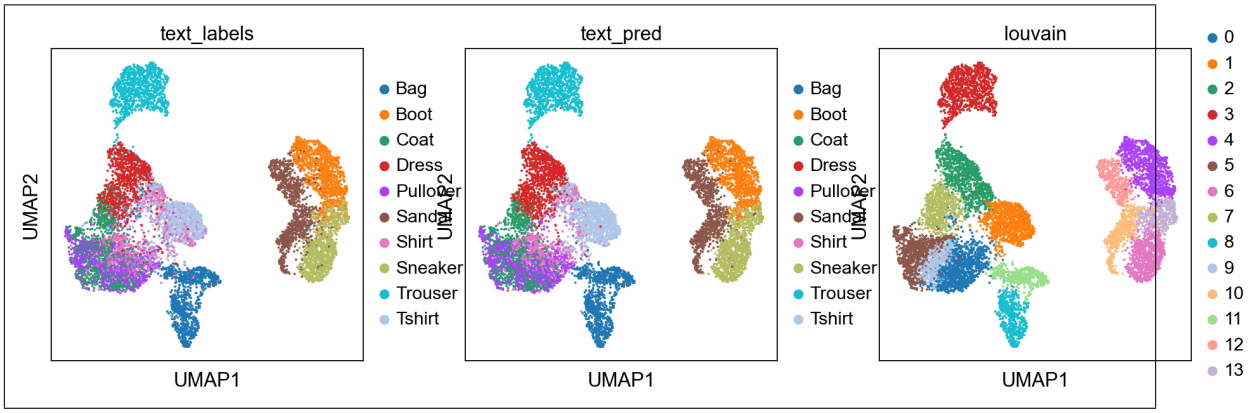


Figure 6.4: FMNIST, semisupervised. UMAP

6.3 Combination with Louvain clustering

Louvain clustering achieves accuracy of 0.71 on both the training and the testing sets. With supervised training c∗GMΔVÆ achieved 0.714 accuracy, an insignificant difference from Louvain. With further unsupervised training rounds it reached 0.725 accuracy. Still below the 0.75 which was achieved with the best model during unsupervised only training but it was only obtained with much trying and tuning. The combination with Louvain is much more consistent as a method.

6.4 Comparison with other models

The Fashion-MNIST repository [3] includes information about the classification accuracy obtained by various models. It is not stated which of the models if at all is unsupervised or semisupervised (for what we see none are). To be fair with the comparison our model was tested in fully supervised mode and achieved 0.9 accuracy. An accuracy of 0.92 was reached with a modified model that uses a kind of deep convolution neural network (RESNET18[5]) in its decoder network. It was done mostly as an anecdote because in this work we don't focus on the network architecture as much as in the probabilistic model. The network architecture unless otherwise specified, is kept fairly simple (2-4 fully connected layers, with batch normalization and dropout). The best classifier listed on the Fashion-MNIST website achieved 0.963 accuracy (WRN-28-10+). It looks like methods that don't use image specific tricks and tweaks (things like convolution network, augmentation by flips, rotations etc …) achieve 0.9 or less accuracy. For example a support vector classifier (SVC), a kind of supervised classifier, is achieved accuracy of 0.897.

In principle it is possible to utilize neural networks that prove to be good classifiers in the Encoder modules of the c∗GMΔVÆ as a drop in replacement ; And networks that include reconstruction modules can be utilized in the c∗GMΔVÆ decoder modules.

Chapter 7

Learning PBMCs

7.1 About scRNASeq

Single-cell-RNA-seq [15] (scRNASeq) is gene expression data on single cell level. The raw data is a matrix of non-negative integers, where rows represent cells and columns represent genes. The integer value represents expression "count"– roughly how many RNA translation copies were captured by the sequencing.

There are are various kinds of noise and distortions which typically afflict these datasets. For example some genes might get zero count in a particular cell even though they are highly expressed in it. There are batch effects, where taking two separate measurements from essentially the same cell population might come out different because of different equipment, or sloppy technicians or what not.

For the above reasons this data requires preprocessing before it can be analysed. A good practical guide for for preprocessing and analysis can be found in the Scnapy [25] documentation, a tool which is extensively used with in this thesis. In short, the data is filtered to get rid of some bad reads, and then it is normalized so that each cell gets an equal amount of total counts. This normalized count data than undergoes $log(1 + x)$ transformation. The result is a dataset that is comparable to other log-normalized datasets because the expression level of a gene per cell is relativizad. The scRNASeq datasets used in this work were taken from the sources of one paper [14] and came already preprocessed and log-normalized.

7.2 a Beginner level dataset for starters

For the initial tests, an "easy" dataset was selected. In this scRNASeq dataset [26] the cells are nicely clustered by cell type. The main challenge with it is that it is imbalanced and some categories are sparse. It has 2623 samples of 7 cell types overall but in particularly one type (dendritic cells) only has 37 samples.

While Louvain clustering performs well: it achieved 0.93–0.94 accuracy and found 9 clusters, c∗GM∆VÆ failed to cluster the smallest cell type (see figure 7.1). With the other types it did ok with accuracy of around 0.92.

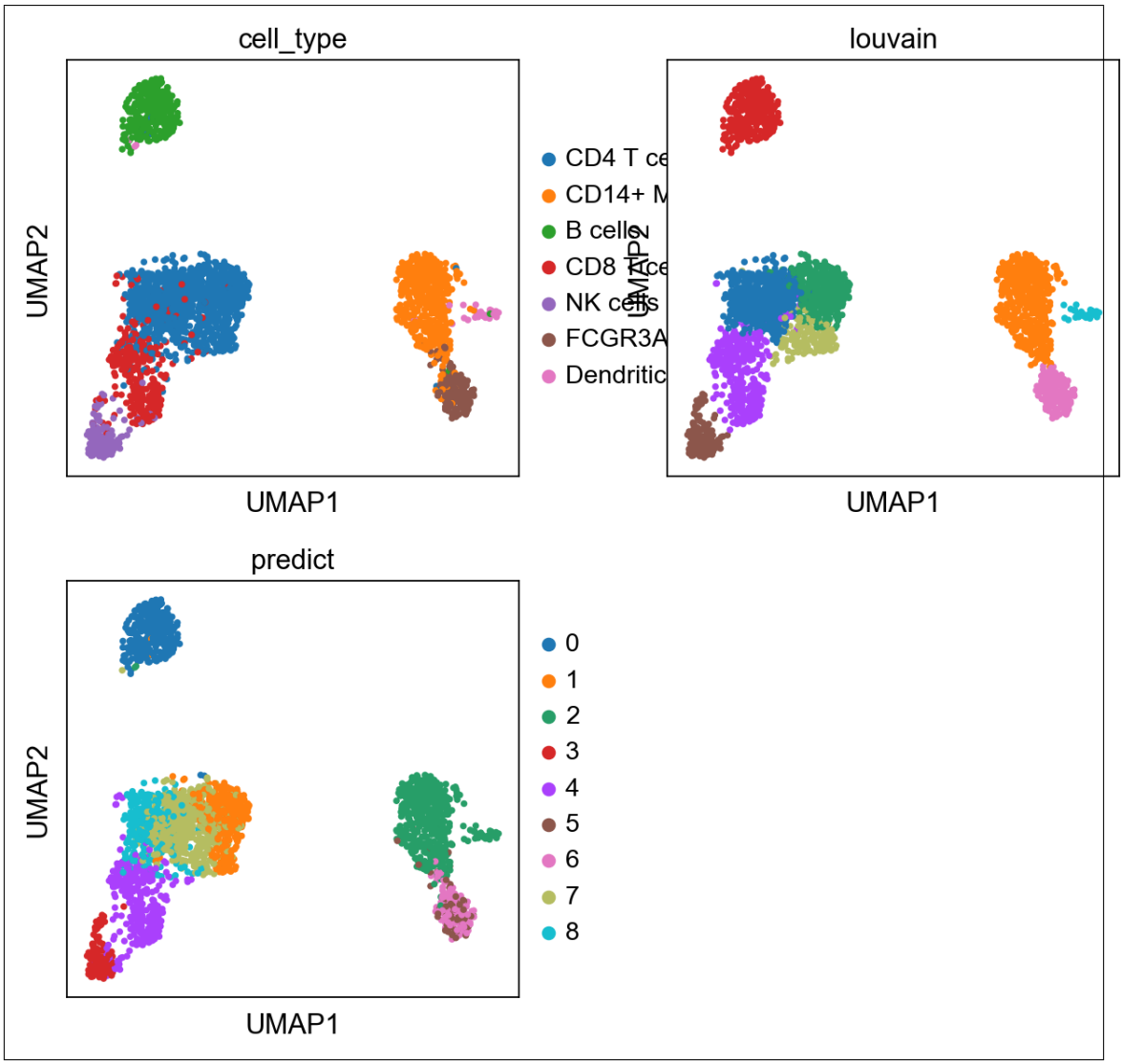


Figure 7.1: Zheng dataset, UMAPs. The main issue with c∗GMΔVÆ is it fails to classify the smallest group (Dendritic) in its own cluster.

In order to confirm that the imbalance and in particular the small sample size of the Dentritic class was the culprit to the misclassification of c∗GMΔVÆ , a rebalanced dataset was generated by selecting 300 samples from each types with repetitions. In addition small Gaussian noise was added. The log-count matrix of the original data set was rescaled, so each gene has near standard normal distribution across the samples. The last step was optional and most preprocessing ”recipes” avoid it. In the course of the testing, rescaling seemed to not improve but rather somewhat worsen the results.

A c∗GMΔVÆ model was trained by taking turns, first on the rebalanced data, then on the original data, repeating several times. It was able to achieve 0.93 accuracy consistently and probably the method can be a little bit improved by tuning the model and the balancing preprocessing. In particular it appears that adding the small noise to the rebalanced data helped the model to learn all the cell types with fewer components. Figure 7.2 visualize with UMAPs the results of this rebalancing method. It goes without saying that such rebalancing process is only possible if the labels are known. It would still be possible to balance the data by using louvain clusters, and a possible another fix would be to just generate more noisy replicates of all the cells without accessing the label information. In

this case we would still have an imbalanced dataset but there will be sufficient quantity from every cell type.

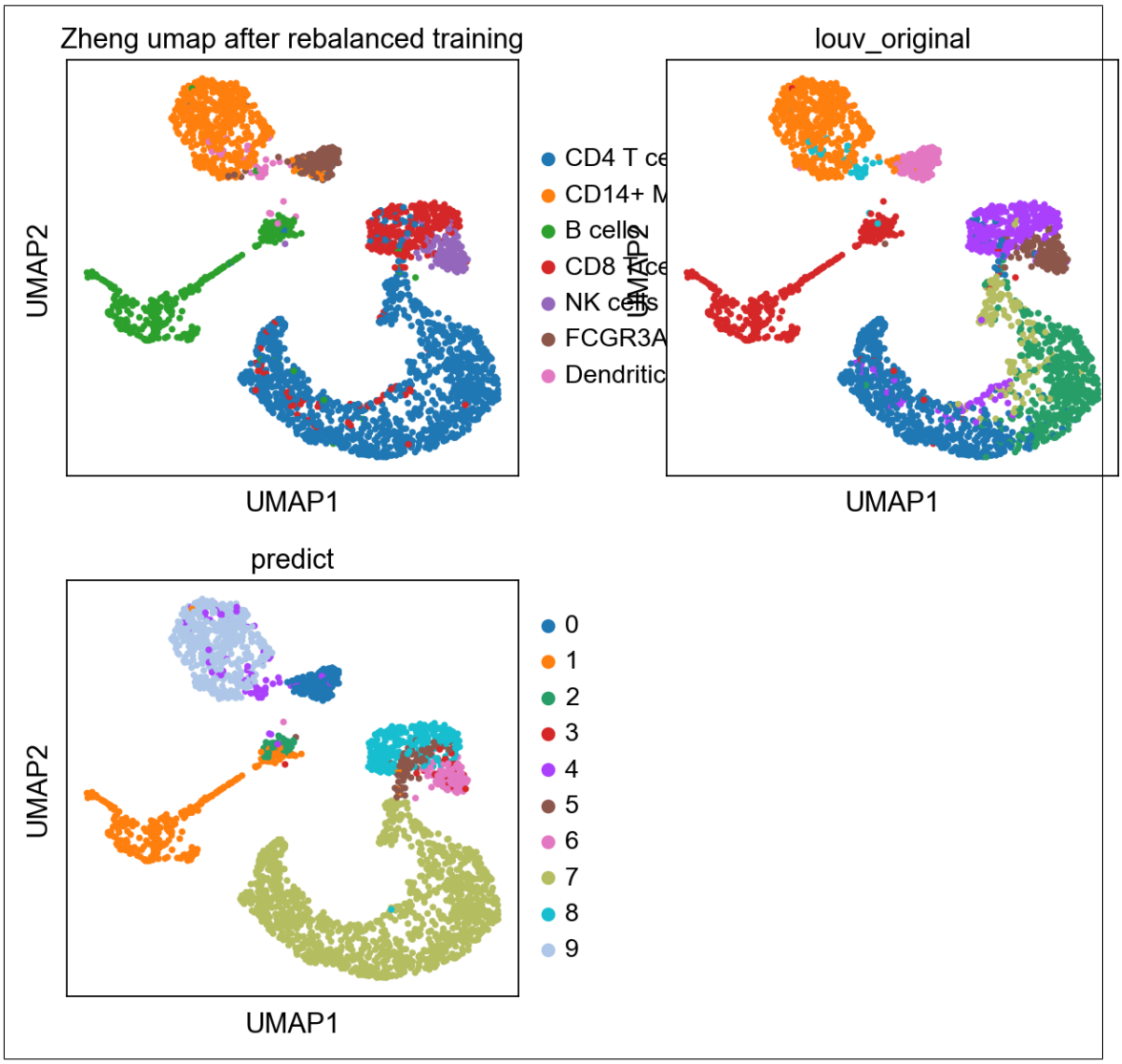


Figure 7.2: Zheng dataset, UMAPs of the latent space after training on the rebalanced and the original data set.

Chapter 8

Larger but messier PBMCs data

This chapter involves an scRNASeq dataset [7] of PBMCs (peripheral blood mononuclear cells) of the same 7 cell types as in Zheng dataset. However the cells in this dataset comes from two groups: control and treatment. The cells in the treatment group have been exposed to interferon beta (INF-β). Moreover, the Kang dataset is much more challenging than the Zheng dataset because the cell types are not nicely clustered as they are in Zheng.

The train/validation split which is used is the one given by [14].

8.1 Unsupervised learning of the control subset

In this section just the control subset of the Kang dataset is used. We first test unsupervised learning of c∗GMΔVÆ . Then we try to see if pre-learning the Zheng dataset with supervised learning can help classify the Kang dataset.

When Louvain clustering with default values was performed on the Kang train data, it missed the CD8 cell type (which was merged with NK type) as figure 8.1 (top left vs bottom) is showing. The same issue occurred in the smaller validation subset. The accuracy of the Louvain clustering was 0.81 in the train subset and 0.78 in the validation subset.

Kang dataset isn't as unbalanced as the Zheng one, and it is larger. Every cell type has over 400 samples from each condition at least.

c∗GMΔVÆ with 21 components achieved 0.78 accuracy on both the train set and the validation set. However it also captured all 7 cell types in its clusters unlike Louvain which missed the CD8.

Figure 8.1 shows the UMAP on the pca space, which lets us compare the original Louvain clustering with the c∗GMΔVÆ classification. The situation with Louvain cluster on the latent space and UMAP of the latent space is essentially the same so it is not shown here.

Perhaps the c∗GMΔVÆ performed essentially as well as Louvain and was even able to capture the cell type missed by Louvain because that the Dataset is larger and better balanced than the Zheng data. The sample size per cell type in the train set is in the

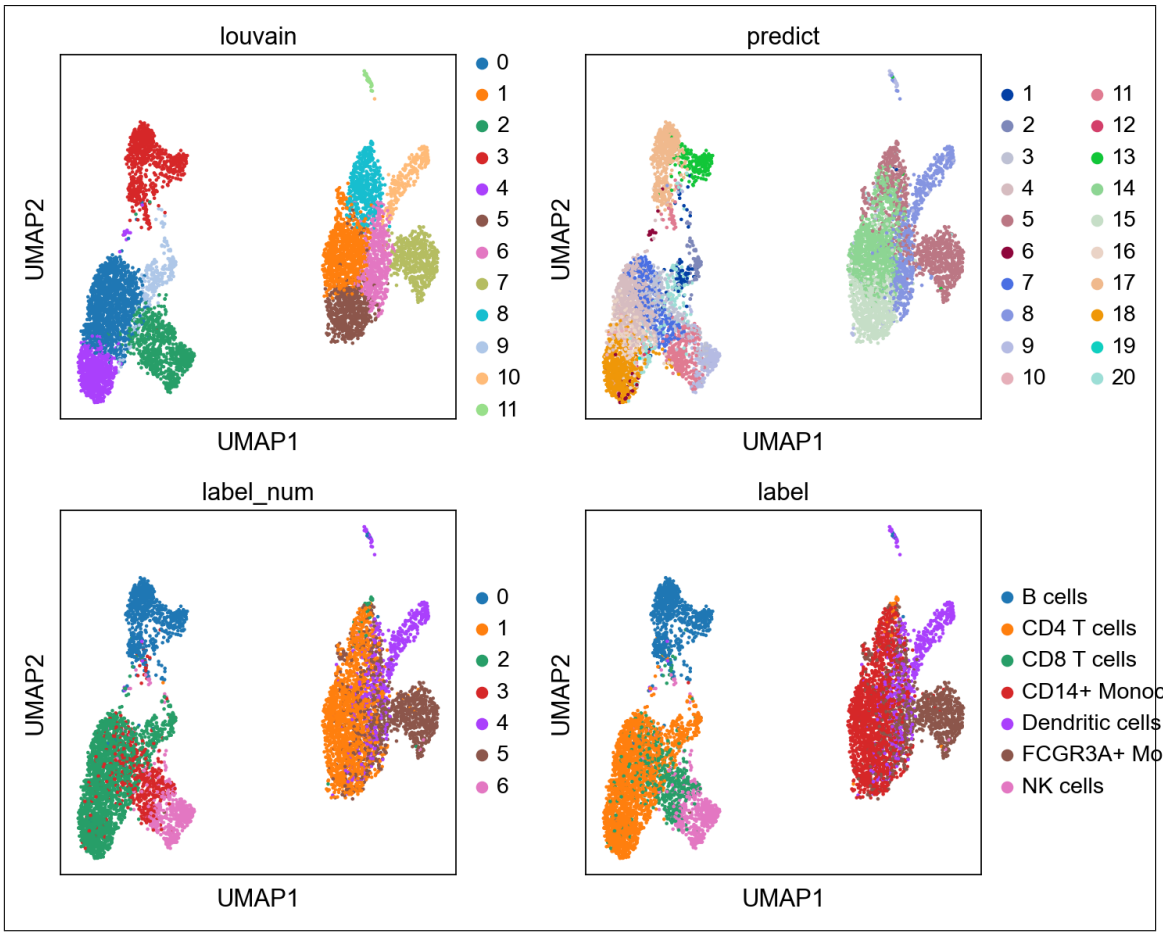


Figure 8.1: UMAP of the Kang train data. It is interesting that the c∗GMΔVÆ ”clusters” can be fuzzy, unlike Louvain clusters which tend to have clear borders.

range 500–2000 unlike the situation in the Zheng data where one cell type has 37 samples and another over 2000.

8.2 Transfer learning

The Zheng and the (control) Kang datasets both represent the same PBMCs However the former is a much nicer dataset. The idea here is to do semi-supervised training, where a re-balanced Zheng dataset is used as the labeled training subset, and the Kang training dataset is the unlabeled subset.

As mentioned in the previous section, Louvain clustering performs ok-ish on the Kang in terms of overall accuaracy (0.78-0.8) but it missed completely one of the classes (CD14).

Figure 8.2 shows the UMAP of the Zheng data. There are 8 Louvain clusters (not sure why in the previous chapter there were 9), and they capture fairly accurately (0.92) the real classes. Moreover the Louvain clusters have well defined borders whereas the true labels are somewhat mixed in with each other (CD8 spots inside the CD4 blob). Two variations for the labeled subset were tried: the true labels and the Louvain labels. With the first variation, it couldn't consistently capture all the classes. The second variation turned out better. It must be that the extra mixture component (there are more Louvain cluster than cell types) helps the model to capture all the ground truth classes.

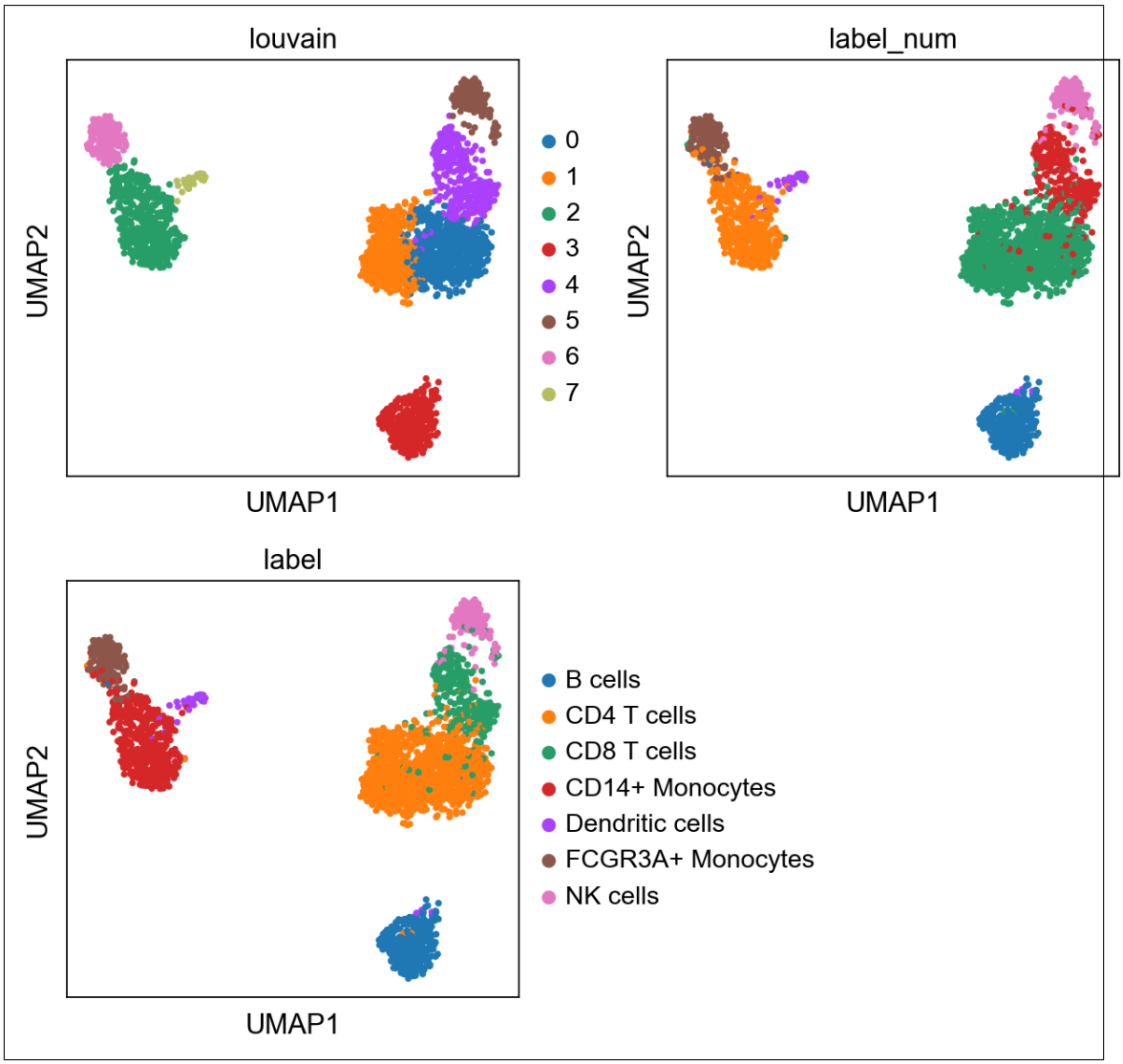


Figure 8.2: UMAP of the Zheng dataset.

Figure 8.3 shows the result of the semisupervised training, where Zheng Louvain clusters were used as the labels. Technically speaking this was a form of unsupervised clustering in the sense that the use the label information wasn't used in the training at all. The accuracy was 0.81 so still not great but edging out the unsupervised training from the previous section. Moreover a minimal overclustering (just 1 extra component) was used and I think the UMAP of the latent space at least looks better. The main issue is with component 4 (predicting CD14) and component 4 (predicting CD8). It is also possible that the accuracy of the provided cell typing for the Kang dataset is itself not very good. It is also interesting that the Louvain clustering of the latent space is very similar to the prediction, except that it unites components 4 and 5 of the model, and consequently missed the CD8 class.

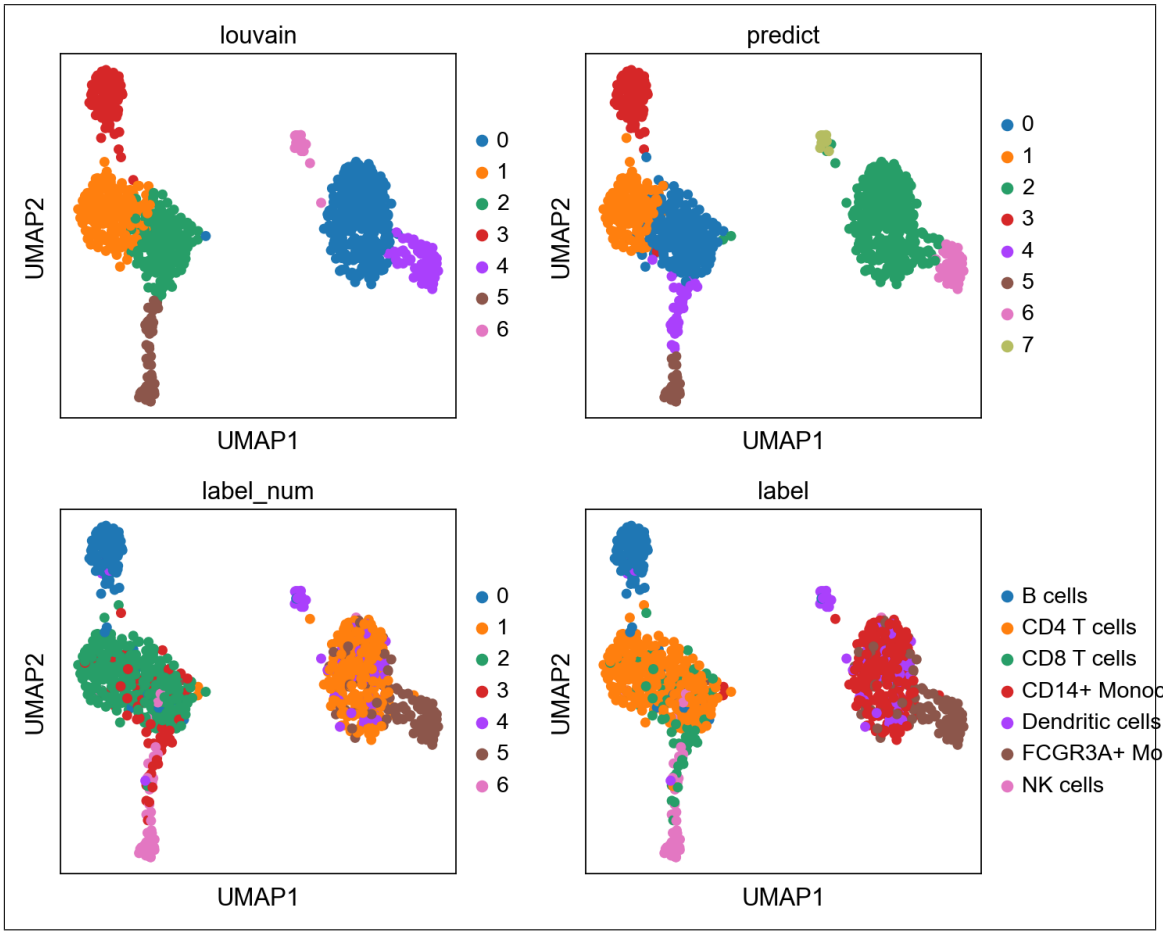


Figure 8.3: UMAP of the latent space of the Kang validation subset (the holdout).

Chapter 9

Testing (conditional) c∗GM∆VÆ on toy data

The conditional variation of c∗GM∆VÆ 4.5 is the most complicated one which was specified in this work. It was decided to first test it on a toy dataset to see if it was worse its salt otherwise why bother with it any further.

The toy dataset is a synthetic 5-class 2-conditions 16-dimensional Gaussian "blobs". Every class at a specific condition is a Gaussian blob. On the same class, the difference in condition is a small random shift plus some noise. It was specifically intended for the condition difference effect to be smaller than a class switch effect. Figure 9.1 shows the toy dataset (by its 3 major PCA components).

It is interesting to directly observe (rather than 'lensing' via UMAP or PCA) the latent spaces $\mathbf{z}$ and $\mathbf{w}$; So in this instance their dimensions were set to 2. In general these dimensions are hyperparameters of the model and setting them too low can lead to sub-optimal results and also cause problems of numerical stability during training. The numerical stability issue is largely solved by constraining $\mathbf{z}$ and $\mathbf{w}$ to a bounded domain. Without it, it is much more likely to get a Nan/Inf error during training. This constraint was set in all of our testings which are brought in this paper.

9.0.1 (cond.) c∗GM∆VÆ with learned prior

In this model, $\mathbf{w}$ has a learned prior $(\mathbf{w}|c)$ which encoder $q(\mathbf{w}|\mathbf{x}, c)$ tries to stay close to. With this setup, we expect that in the $\mathbf{w}$ space of the encoder, we'll see some separation between the two conditions. Within the same condition, we expect to see the classes roughly normally distributed, since each $\mathbf{w}$ as input decodes to any of the classes in the $\mathbf{z}$ space (then to be selected by the y variable for the reconstruction.

On the $\mathbf{z}$ space, we expect to see clustering based on the classes (assuming the model will pick the correct classes for its mixture components). There may also be a visible condition effect within a class on the $\mathbf{z}$ space.

Figure 9.2 shows the means of the encoded $\mathbf{w}$-distribution. We clearly see the two clusters for the "ctrl" and "trmtmnt" conditions. Within each condition, there is some clustering of the classes too. However if we sample randomly from this distribution (9.3,

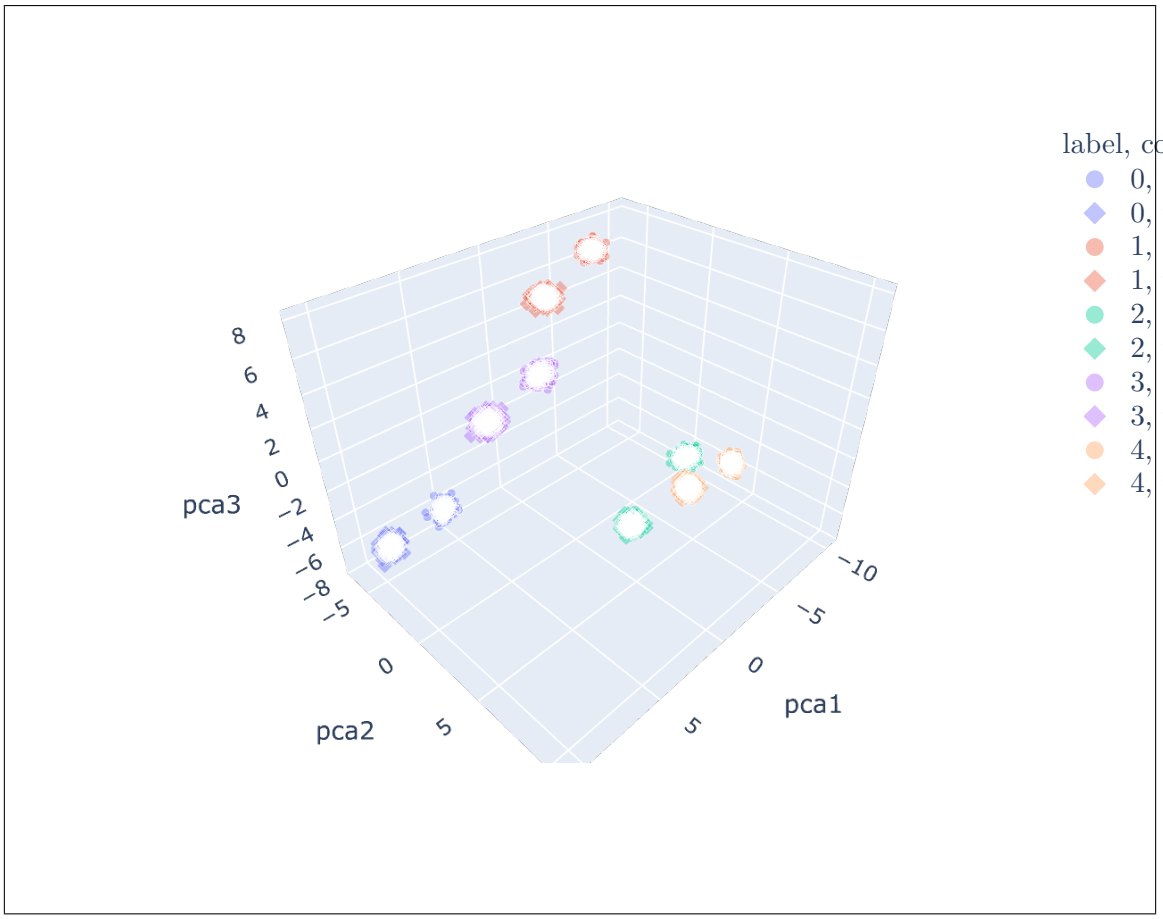


Figure 9.1: The toy dataset, showing a 3d plot of its 3 major PCA components.

it is hard to distinguish any pattern. Everything (class and condition) appears mixed together.

A perfect accuracy (1.0) was reached with unsupervised training. The model was defined with the same number of mixture components as there were classes and it captured every class in its own component with 1.0 accuracy. It suggests that the underlying probabilistic model of c∗GMΔVÆ really matches the distribution of the dataset.

Move on to the **z** space.

We see that in figure 9.4 the means, and in figure9.5 (random samples) that each mixture captures its own class of the dataset. They are completely separated. When looking at the means plot, within a class, we see some hardly visible subdivision based on conditions. Ideally we may want a model that doesn't encode condition-difference into the **z** space, because such a model suggests that the decoder is able to reconstruct from the same input in **z**, A good thing if we want to use such a model to predict treatment effect.

9.0.2 c∗GMΔVÆ with fixed standard normal prior

In this model, **w** doesn't learn a prior but is given a fixed standard normal prior $p(\mathbf{w})$ regardless of condition. The encoder $q(\mathbf{w}|\mathbf{x}, c)$ tries to stay close to $p(\mathbf{w})$. With this setup, we expect that in the **w** space of the encoder will mix the two conditions together. This can be a good thing for example, for batch effect reduction.

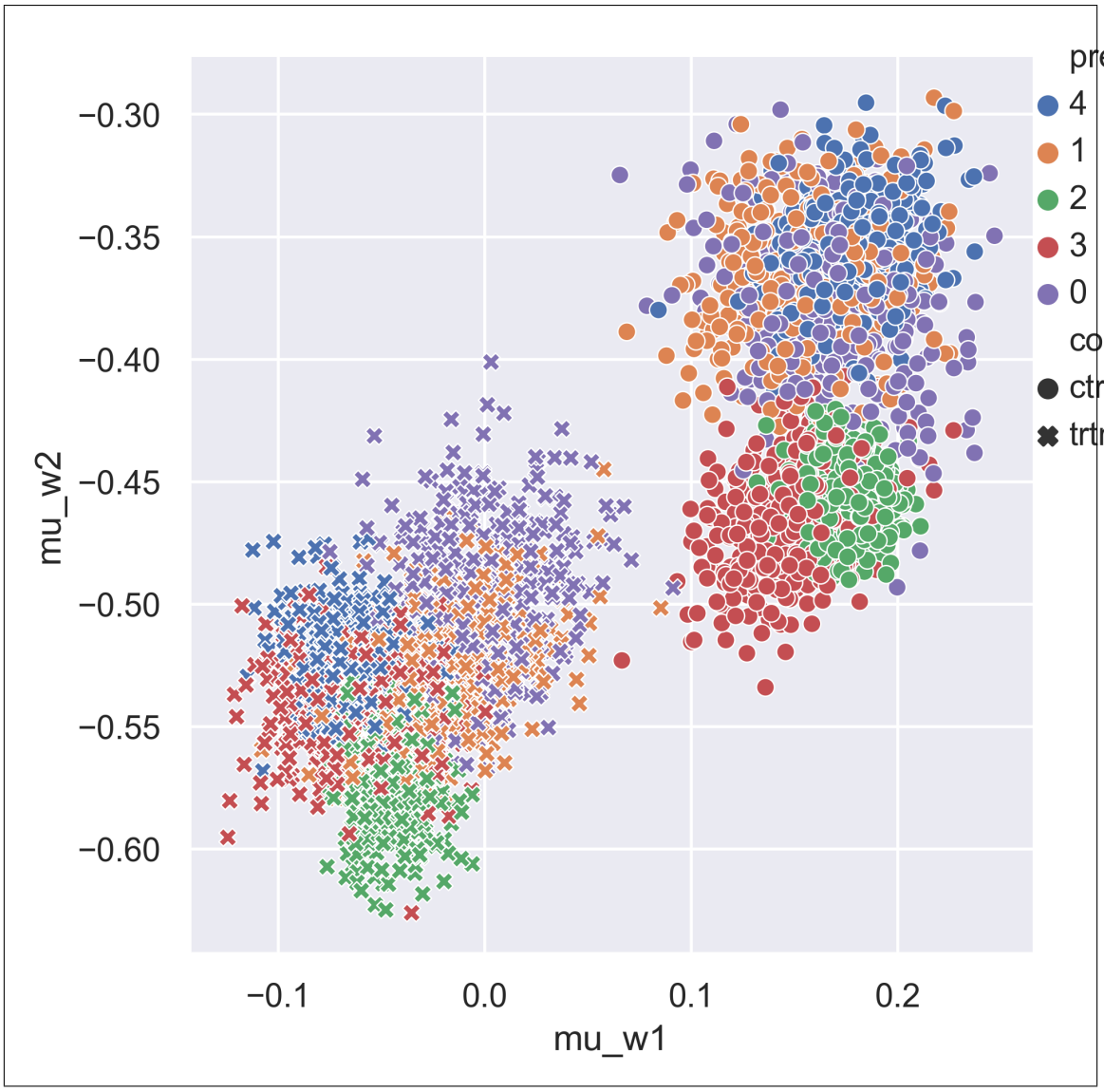


Figure 9.2: The encoded means of the **w** space, learned prior case.

On the **z** space, we expect to see clustering based on the classes (assuming the model will pick the correct classes for its mixture components). There may also be a visible condition effect within a class on the **z** space because the condition c is always passed forward as input for the models sub-networks. This time as well perfect accuracy was reached.

In figure 9.6 we see the means of the encoded **w** space. Here we get complete separation to different classes, but in each class-cluster the conditions are tight together, even if somewhat not mixed in perfectly. An interesting contrast to the learned prior case (figure 9.2. Random samples from **w** aren't shown this time because again, it looks the same as in the learned prior case, with everything mixed together.

In figure 9.7 we see the encoded means and in figure 9.8 we see samples of the **z** space. The means look very tight together. There is hardly any difference within the same class for the different conditions, which again, should be what we want to achieve for the purpose of treatment effect prediction and for batch effect reduction.

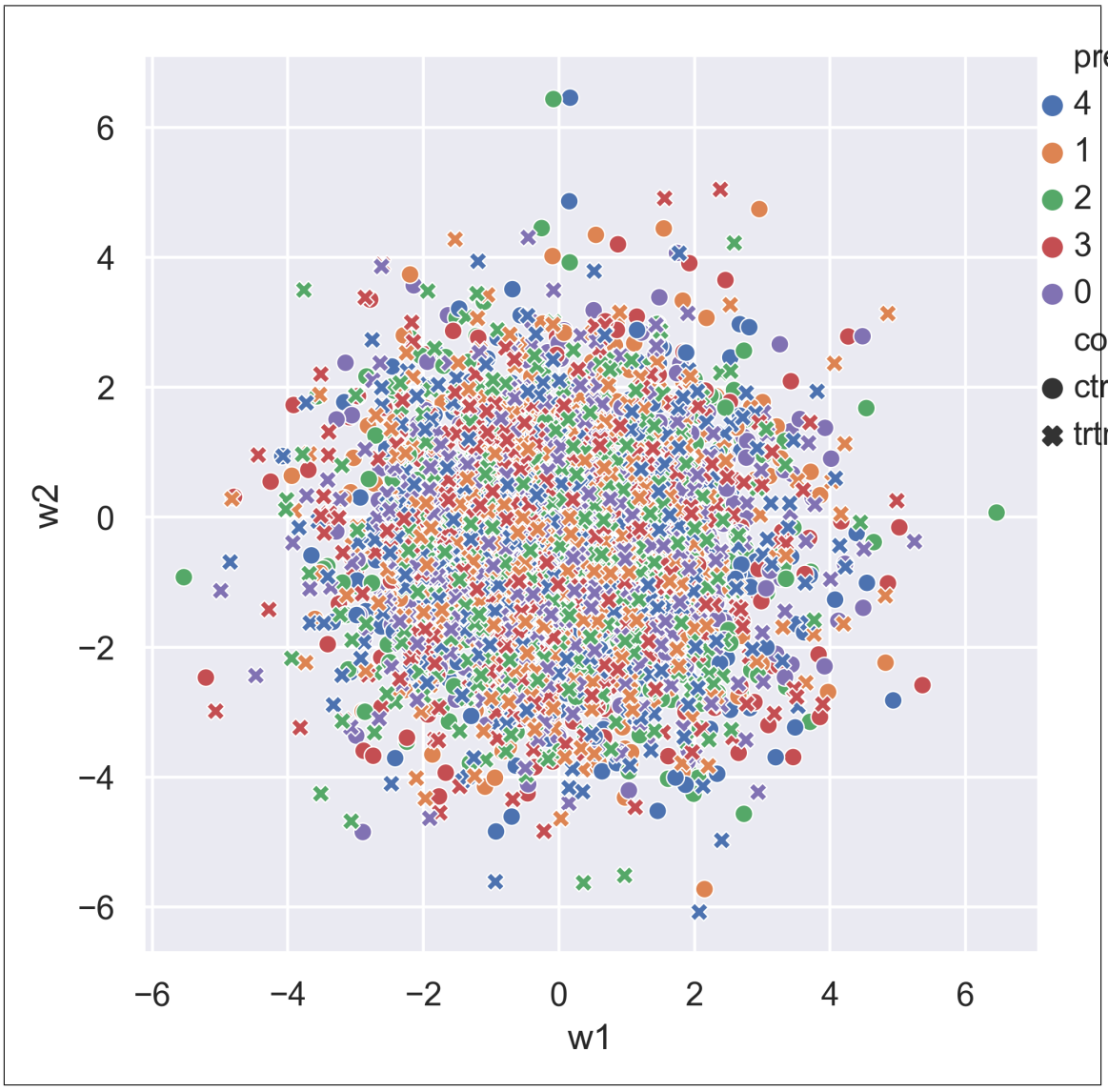


Figure 9.3: Random sampling from the encoded distribution of the $\mathbf{w}$ space.

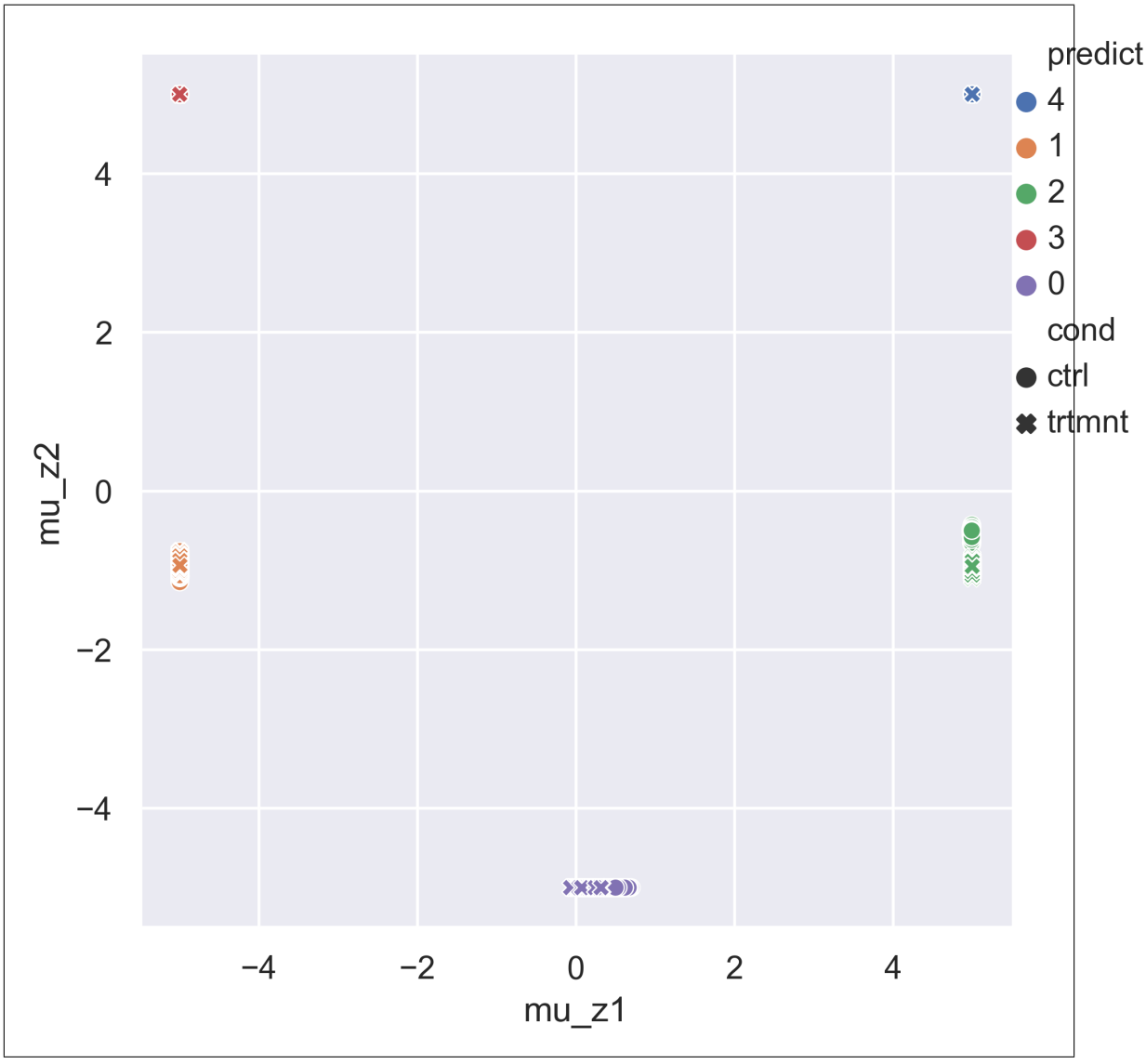


Figure 9.4: The encoded means of the **z** space, learned prior case.

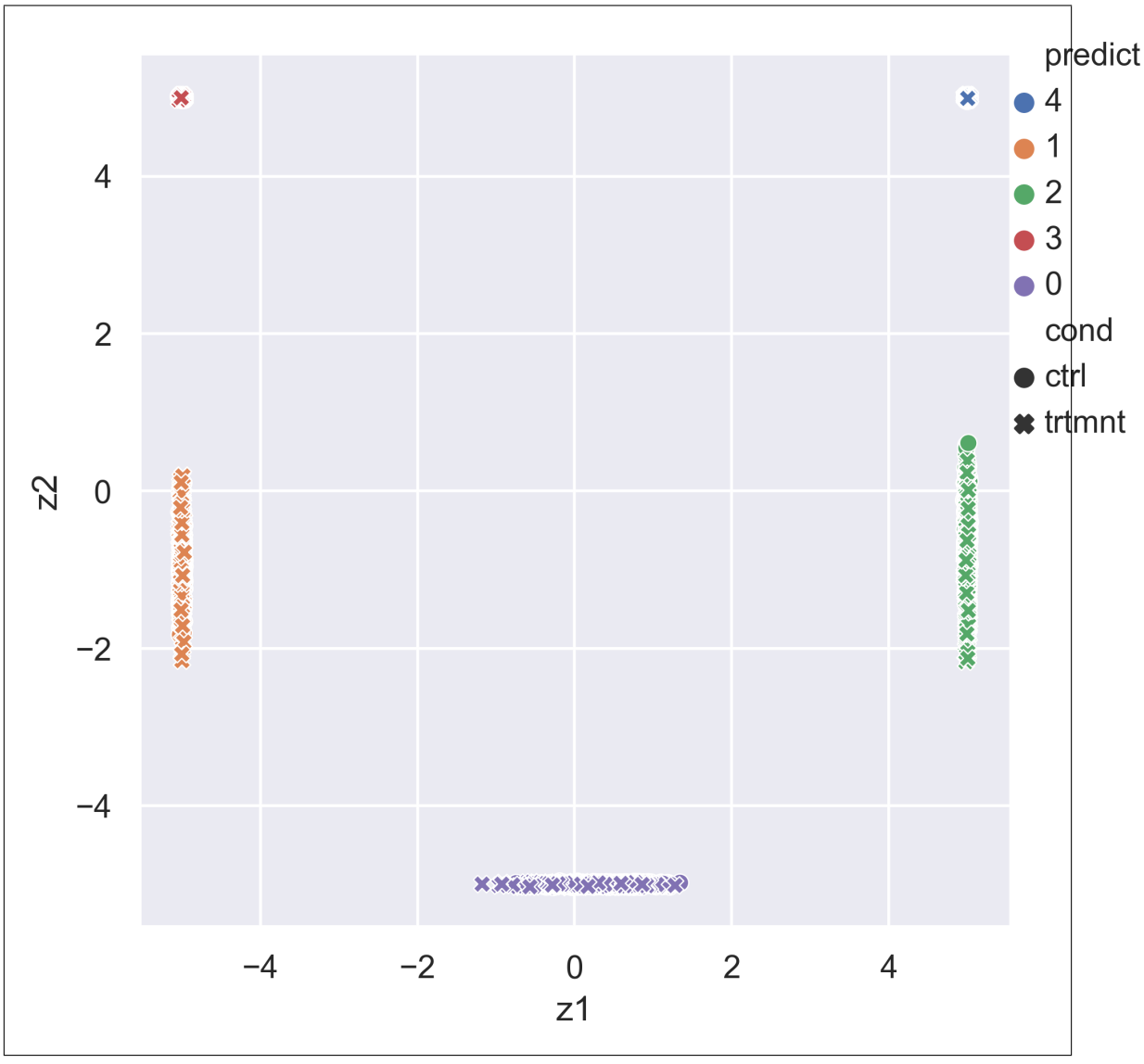


Figure 9.5: Random sampling from the encoded distribution of the **z** space.

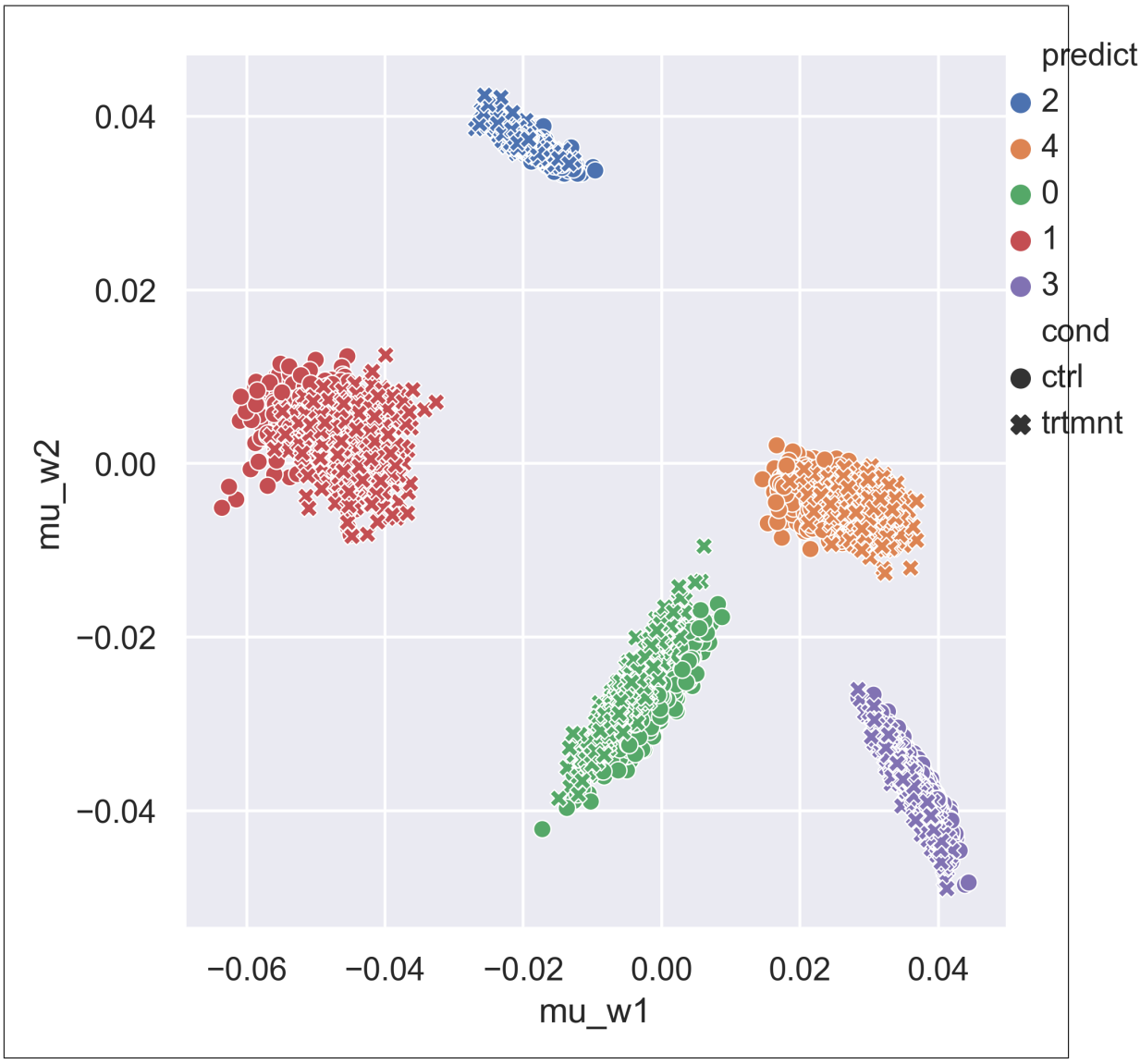


Figure 9.6: The encoded means of the **w** space, fixed standard normal prior case.

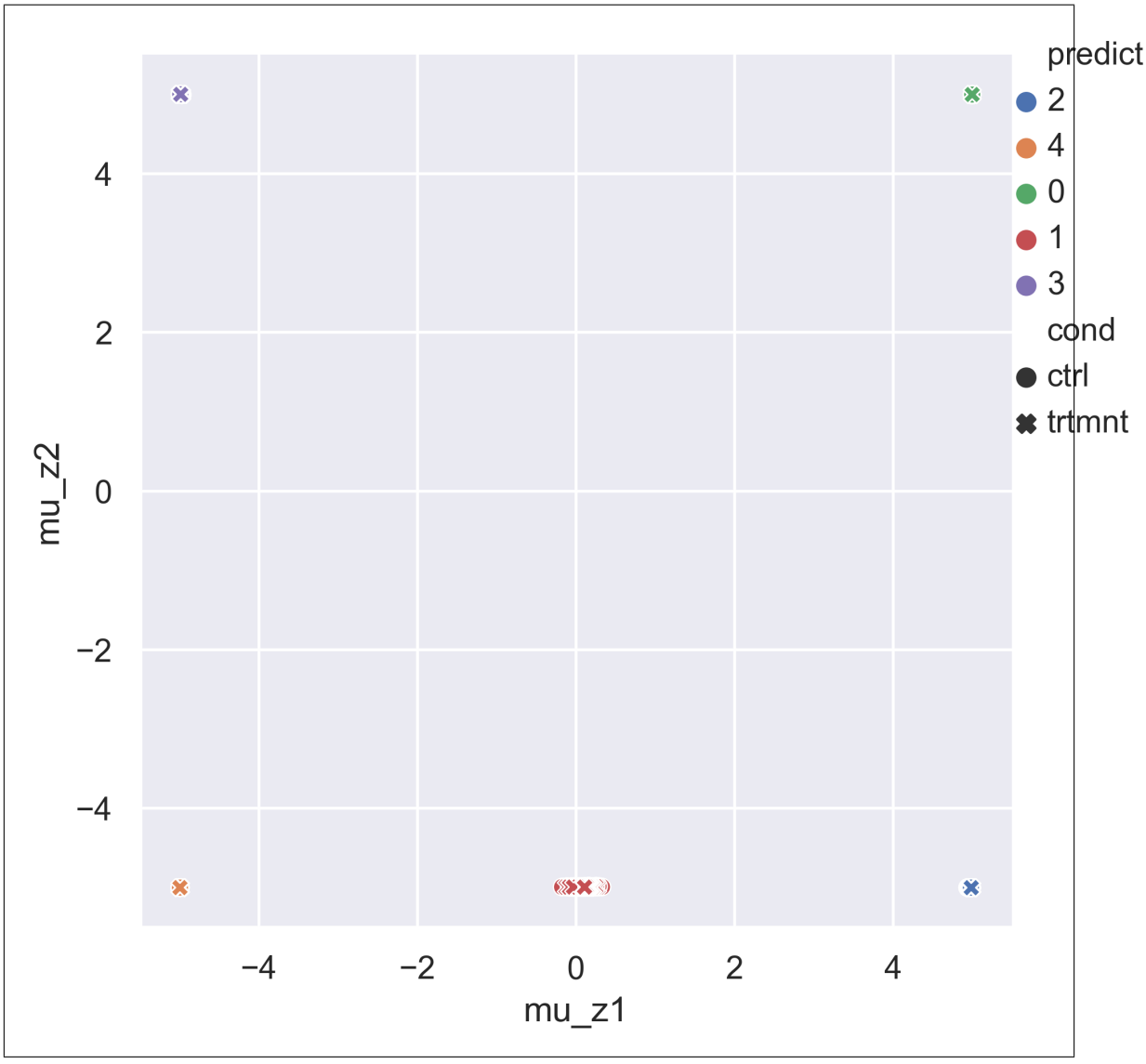


Figure 9.7: The encoded means of the **z** space, fixed standard normal prior case.

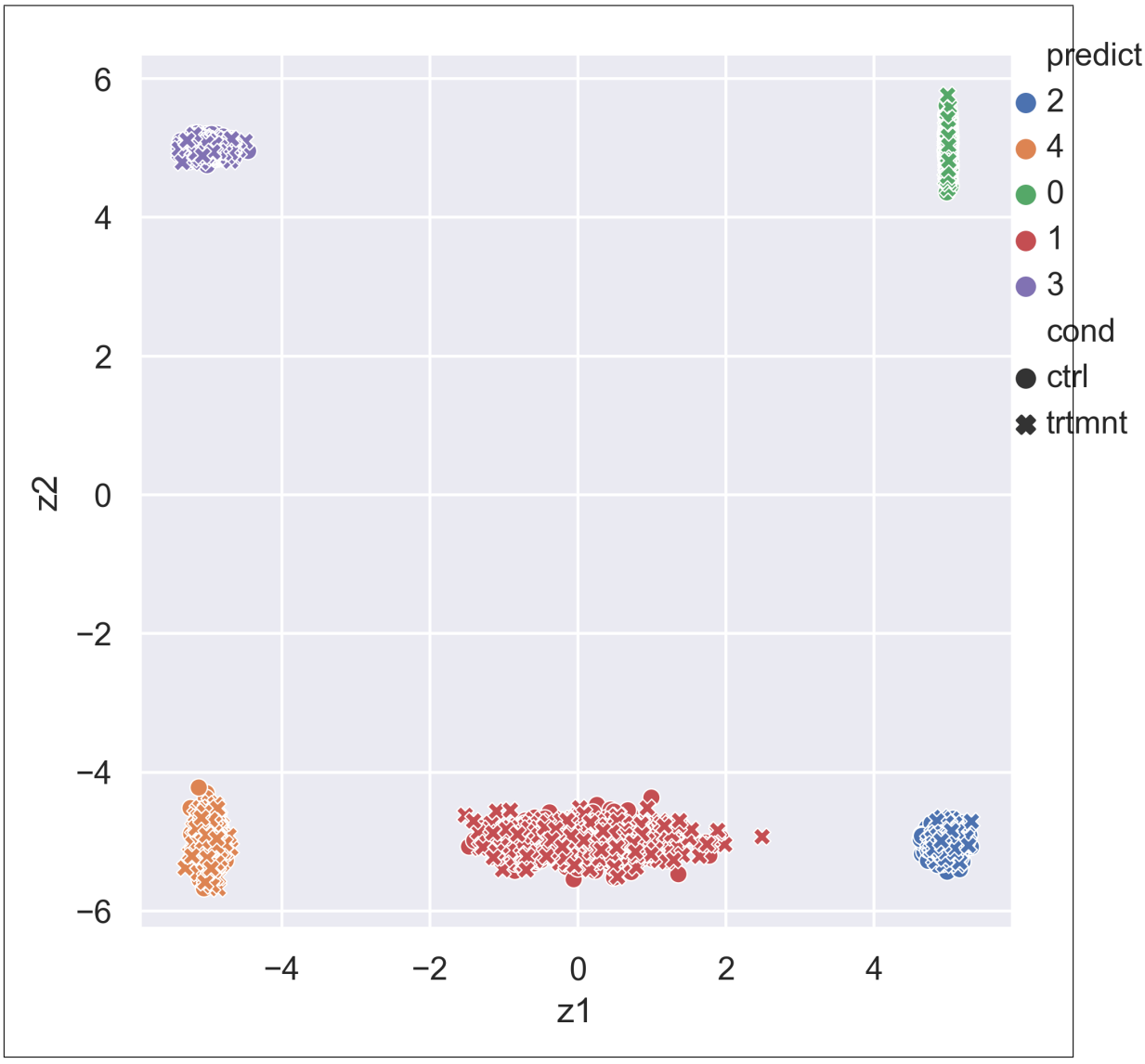


Figure 9.8: Sampling from encoded **z** space, fixed standard normal prior case.

Chapter 10

c∗GM∆VÆ on the treatment/control type PBMC dataset

As mentioned the full Kang dataset contains two groups of PBMCs, the control group (used in the previous chapter) and the stimulated group, the latter contains the gene expressions after exposure to INF-β. The c∗GM∆VÆ model may fit well for this data, where a binary condition variable represents control/stimulated and the mixture component of the represent cell types. In particular we aim at obtaining some interesting results from (semi)supervised learning. This dataset is too hard for the model to learn unsupervised, as we have seen in the previous chapter.

Again the same normalization and the same training/testing (they called it validation) split which was provided by [14] were used. All the training was done on the training subset, and all the plots and evaluations are on the validation subset.

10.1 Unsupervised learning (full Kang dataset)

The Kang dataset has a strong batch effect, where surely part of it is due to the actual treatment effect and the rest (perhaps even the majority) is actual batch effect of taking two separate measurements of different cell populations. Louvain clustering with default usually misses completely class 3 or 4 (CD14 and Dentritic cells). Figure 10.1 shows the situation.

The model used here was c∗GM∆VÆ model without learned prior with latent z and w dimensions of 2 and 28 mixture components. The overall accuracy was 0.78 which isn't very good but at least it was able to pick up all the classes.

Figure 10.2 shows the encoded means in the latent z space. Color indicates class label and node shape indicates condition. We can see that the classes are pretty much clustered together and within a class, the conditions are mixed in together. Its the decoders "job" to map from the latent space back to the reconstruction space and given a condition, it should be able to construct both conditions in the original space from the same input. Figure 10.3 shows the same graph but this time color indicate mixture components and

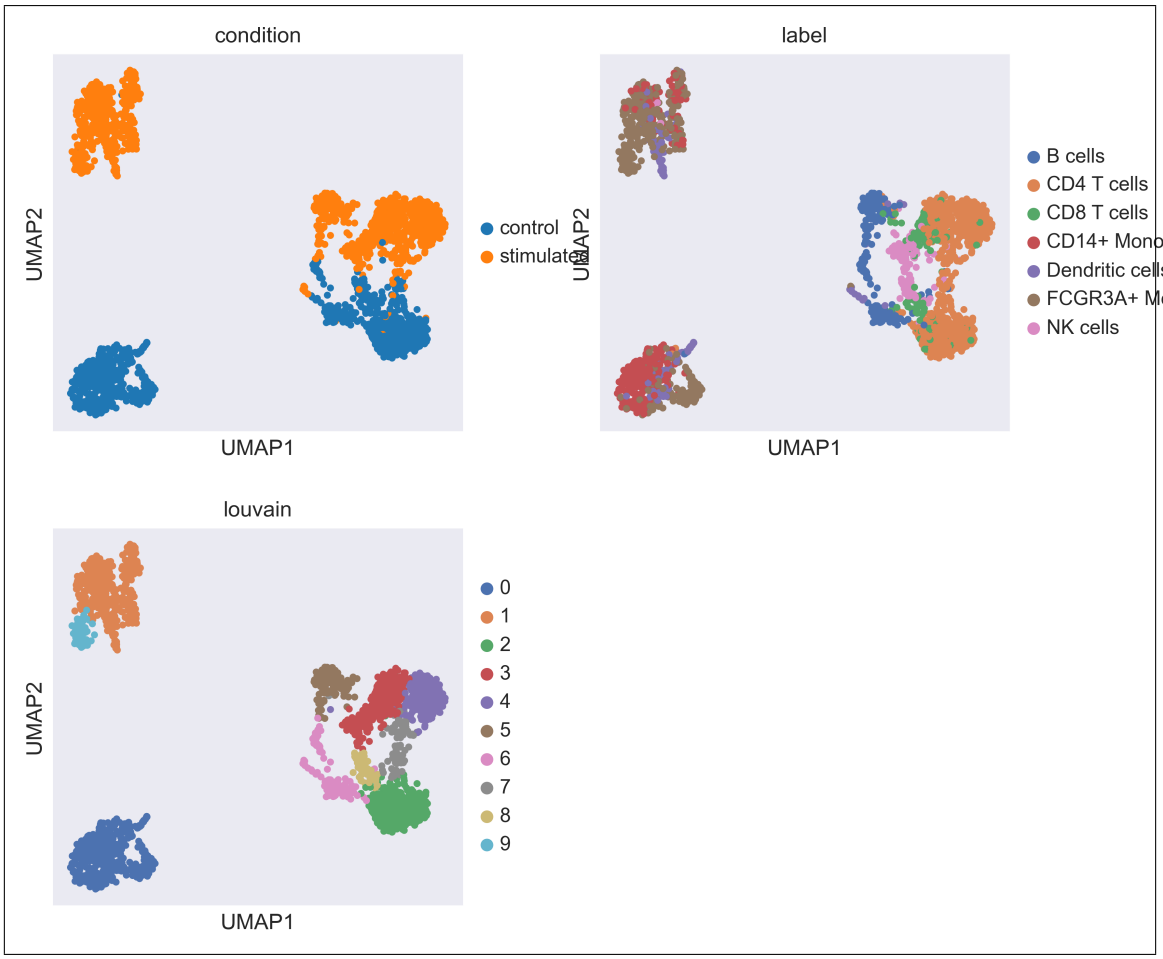


Figure 10.1: UMAP of the Kang validation dataset

node shape indicate actual class label.

We also want to see whats going on in the w space. There we expect everything to really be mixed together. The less clustering and batch effect the better. Figures 10.4 shows the encoded means of the Kang validation dataset. We see that labels and condition are fairly mixed in together. It is interesting that all of the action takes place along a narrow diagonal strip. It looks like the means are encoded pretty much one dimensionally. It suggests that the variance will be working on the other dimension …which is confirmed by figure 10.5. When we take samples from the encoded distribution we get a Jackson Pollock like picture. Everything is mixed in in space.

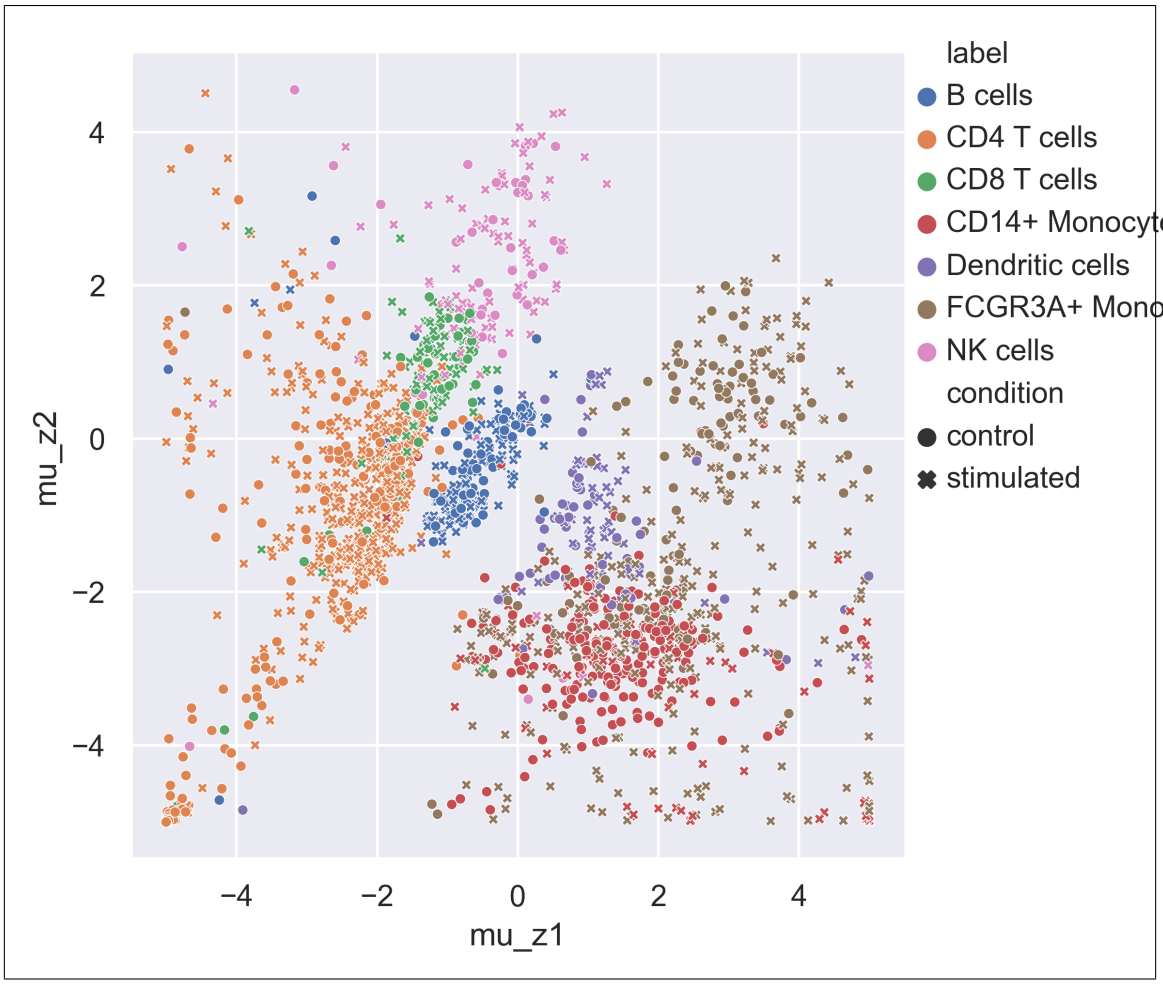


Figure 10.2: Kang data: labels and conditions. The conditions are fairly mixed in.

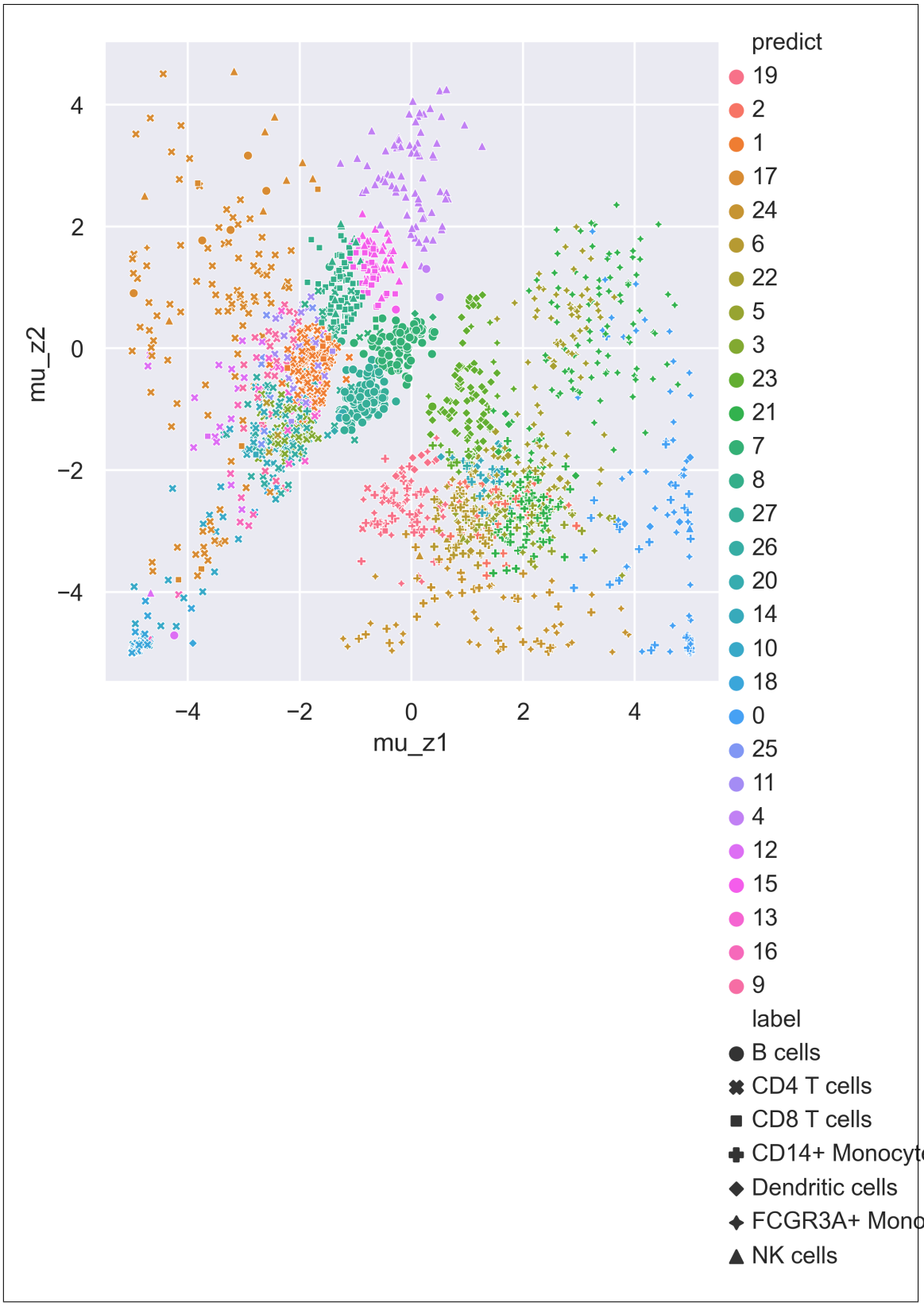


Figure 10.3: Kang dataset. The encoded means of the z latent space. Colors are coded by mixture component ("predict") and node shape are coded by class label.

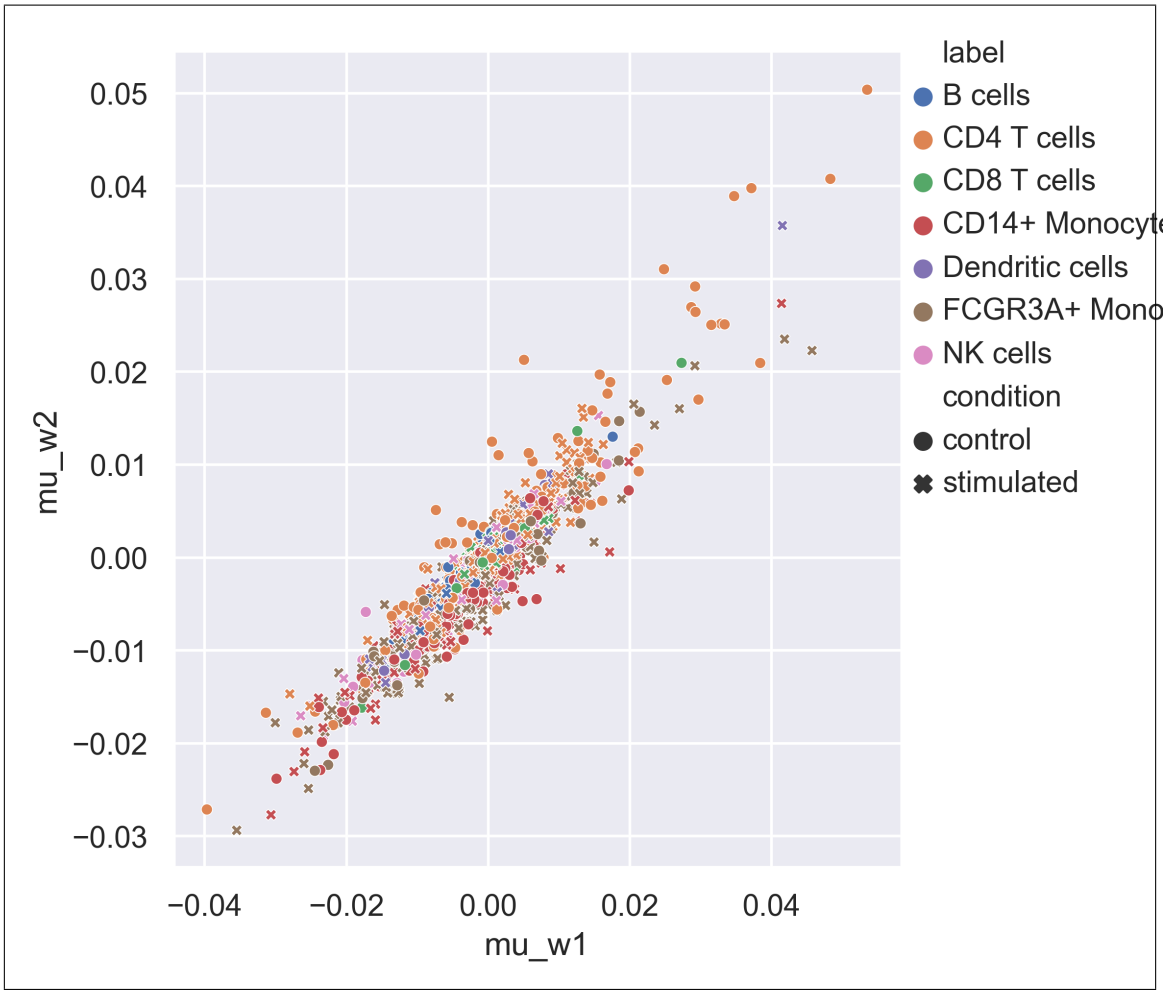


Figure 10.4: The encoded means in the latent w space.

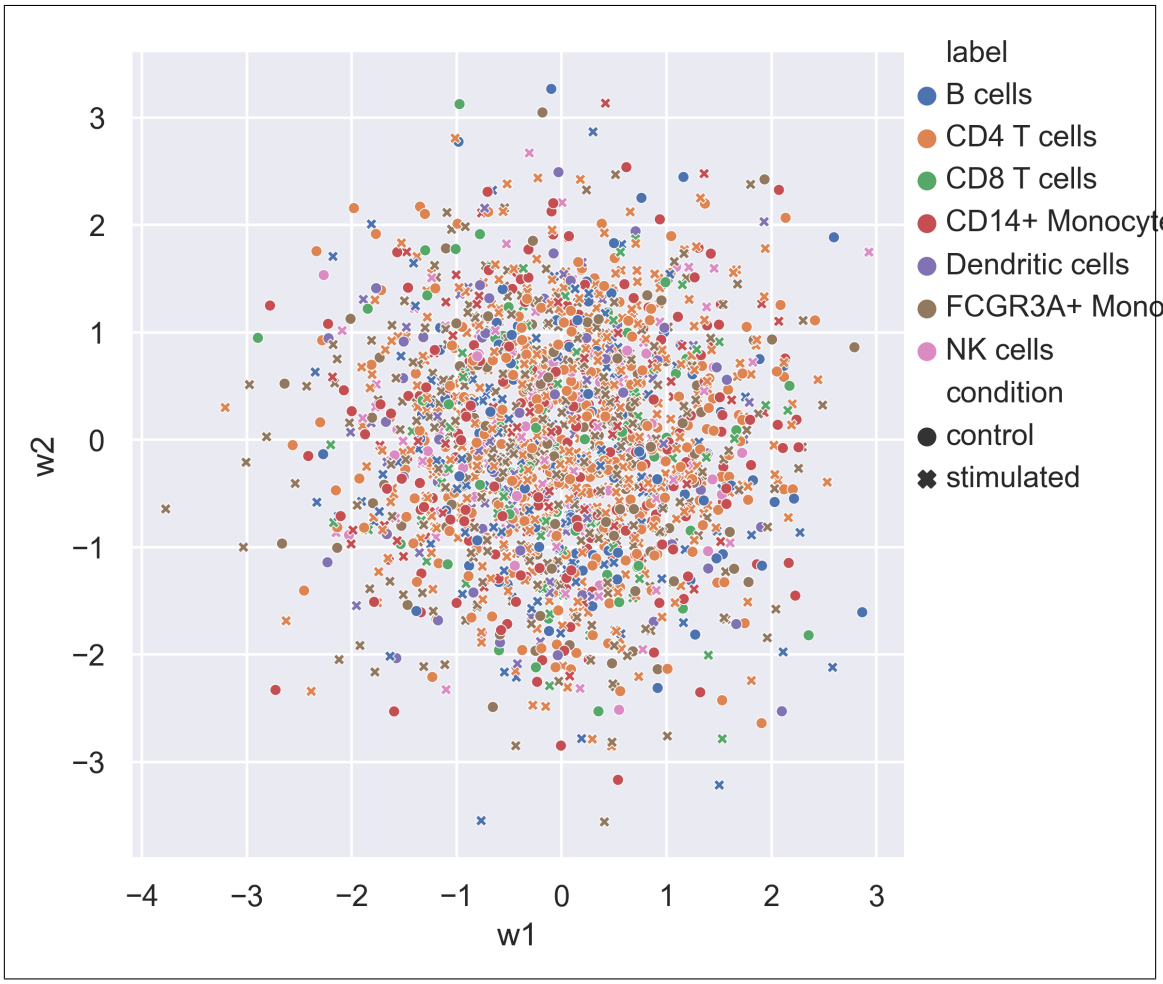


Figure 10.5: The encoded validation subset in the latent w space. This time by random samples from the encoded distribution.

10.2 Supervised training (full Kang dataset)

The main purpose here is to see by supervised training on this dataset c∗GMΔVÆ learns something meaningful and potentially useful. It was specifically intended here to use the version with standard normal prior on the w space (aka no learned prior). We want the conditions and the labels to be deliberately mixed in by the encoder into the w space. Then we can sample from it and from one sample in this latent space, generate every class and in both conditions.

The latent dimensions this time were set at 24 for both z and w because with these higher dimensions it achieved somewhat higher accuracy. The accuracy on the training set reached 0.99 but on the validation subset it dropped to 0.9. Figure 10.6 shows the encoded means in the latent z space by way of PCA. The class labels are fairly if not perfectly clustered. Withing some of the classes there is a clear separation to control and treatment.

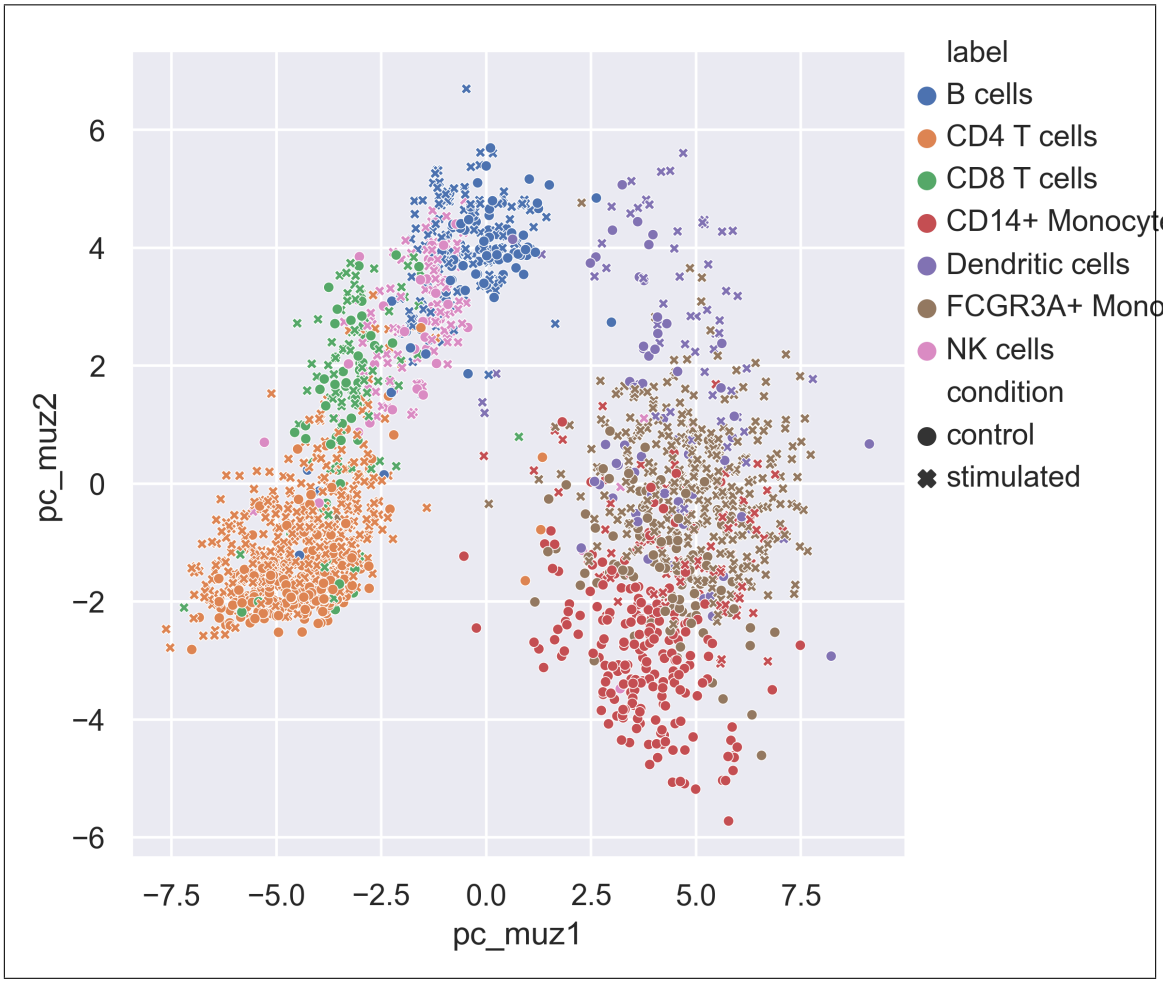


Figure 10.6: The latent z space (encoded means)

When we move to the w space (figure10.8), we see everything is nicely mixed together and normally distributed, which is what we aimed for.

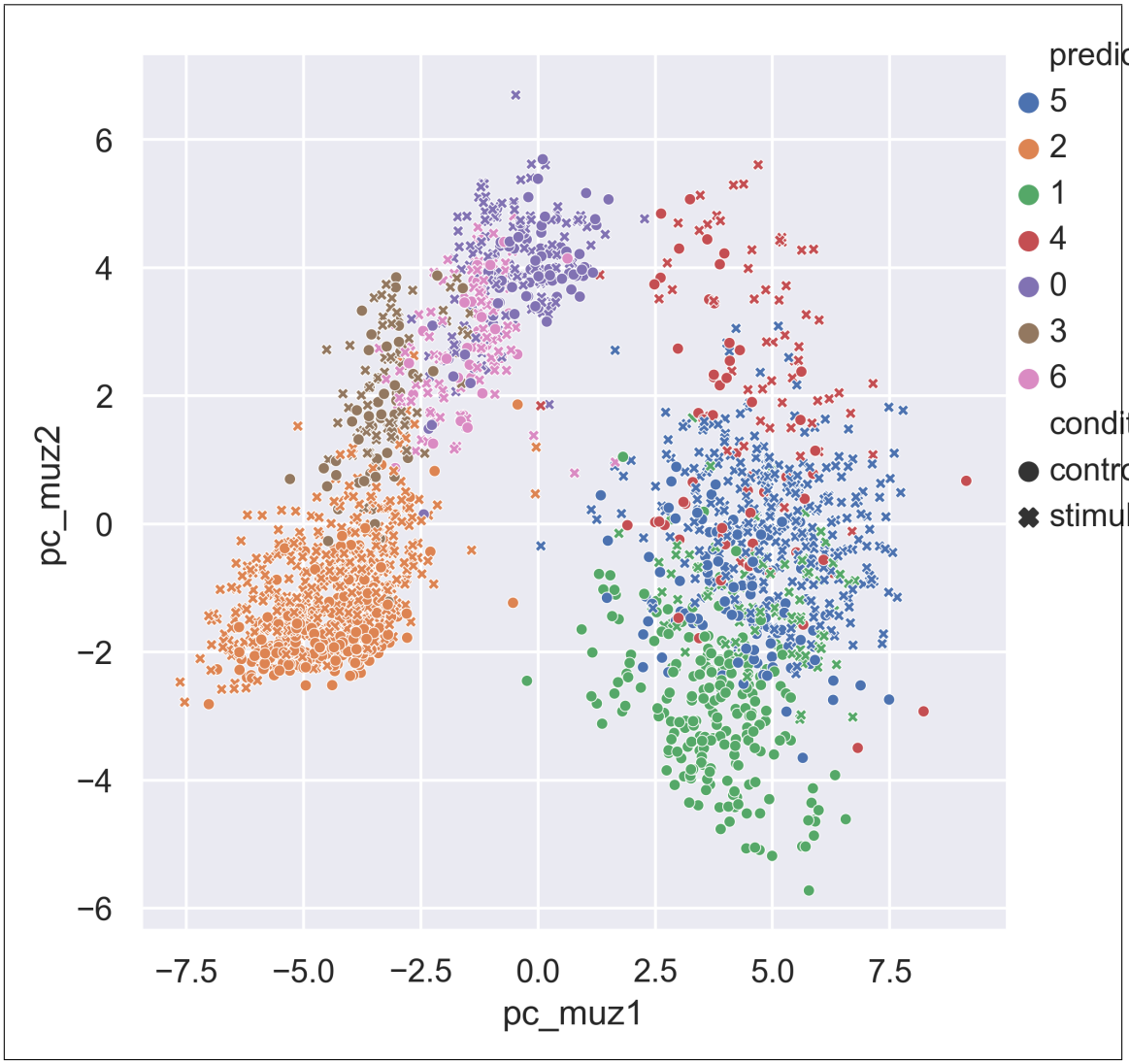


Figure 10.7: The latent z space (encoded means) The mixture's component fit pretty nicely to the real labels (see previous image)

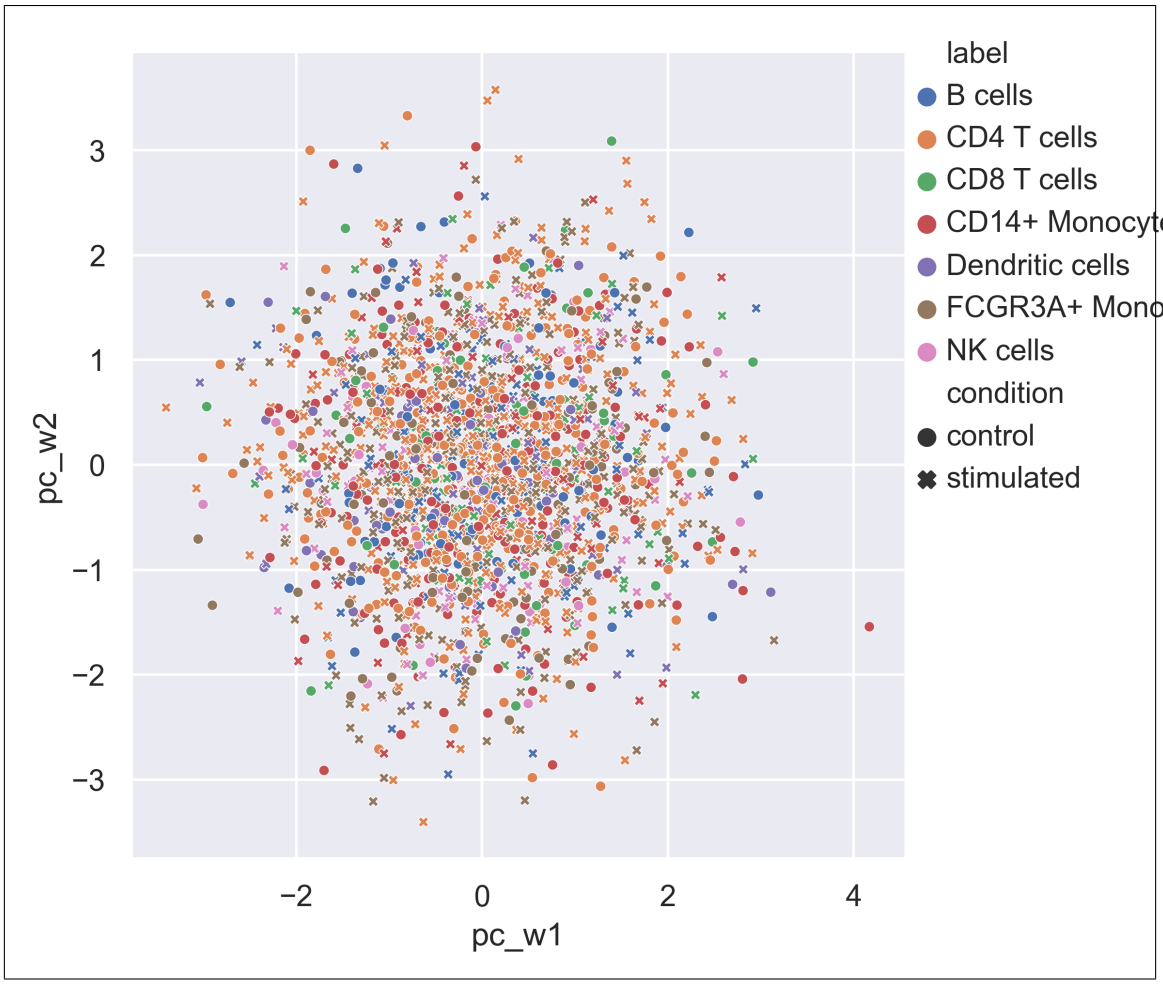


Figure 10.8: The latent w space (sampling)

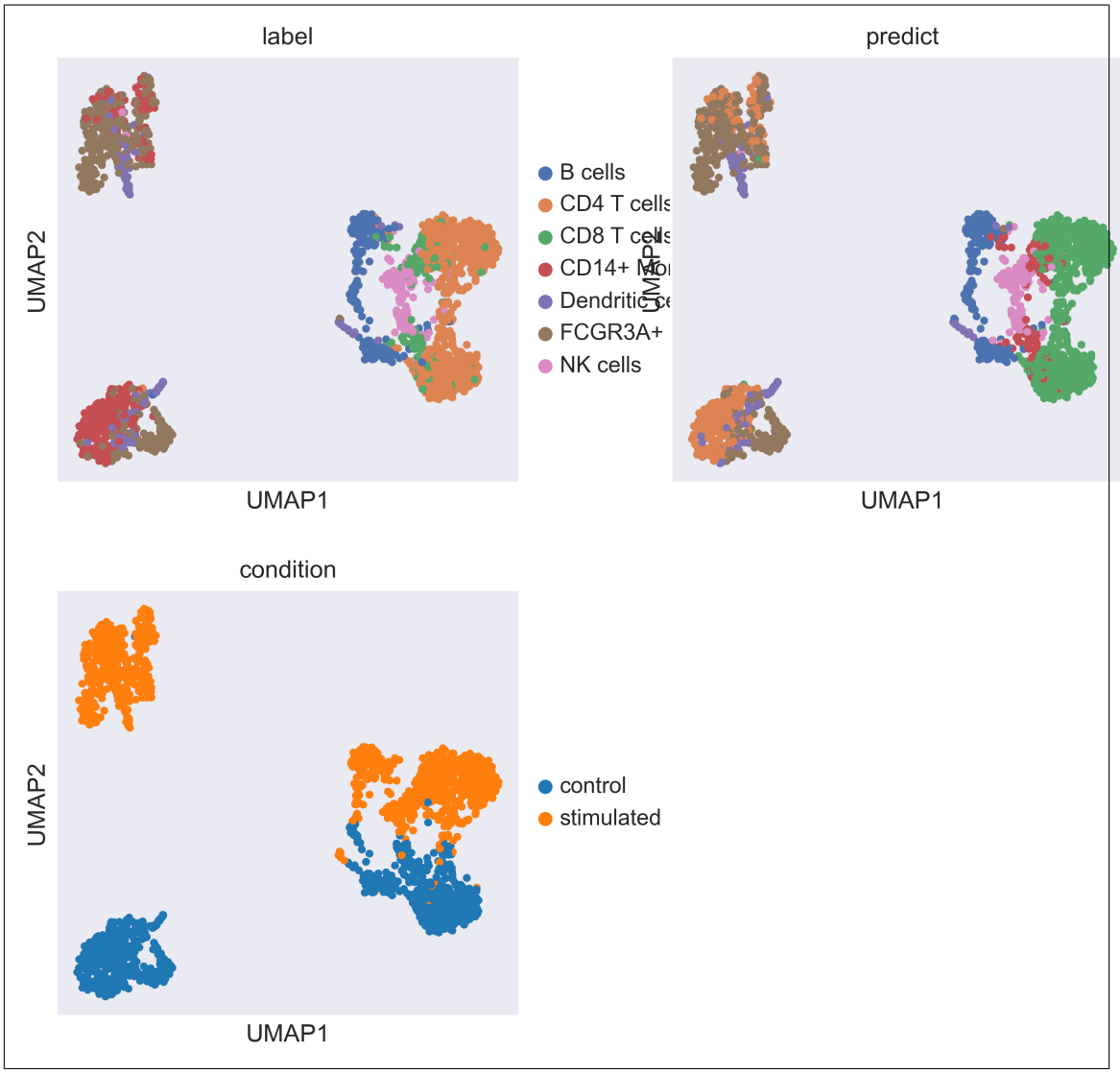


Figure 10.9: UMAP of the Kang validation dataset

10.3 Simulating treatment effect

In this section it is tested whether the supervised trained model can realistically convert cells from the control group to the treatment group. Specifically we take cells from the control group, and encode them into the w latent space, using the real label and condition information.

We take this encoding, keep the correct label but flip the condition and decode it into the z latent space and then back to the input/reconstruction x space. We want that the reconstructed data will look more like the stimulated group than the control group it was mapped from. We also want that cells of the same class will still be clustered together.

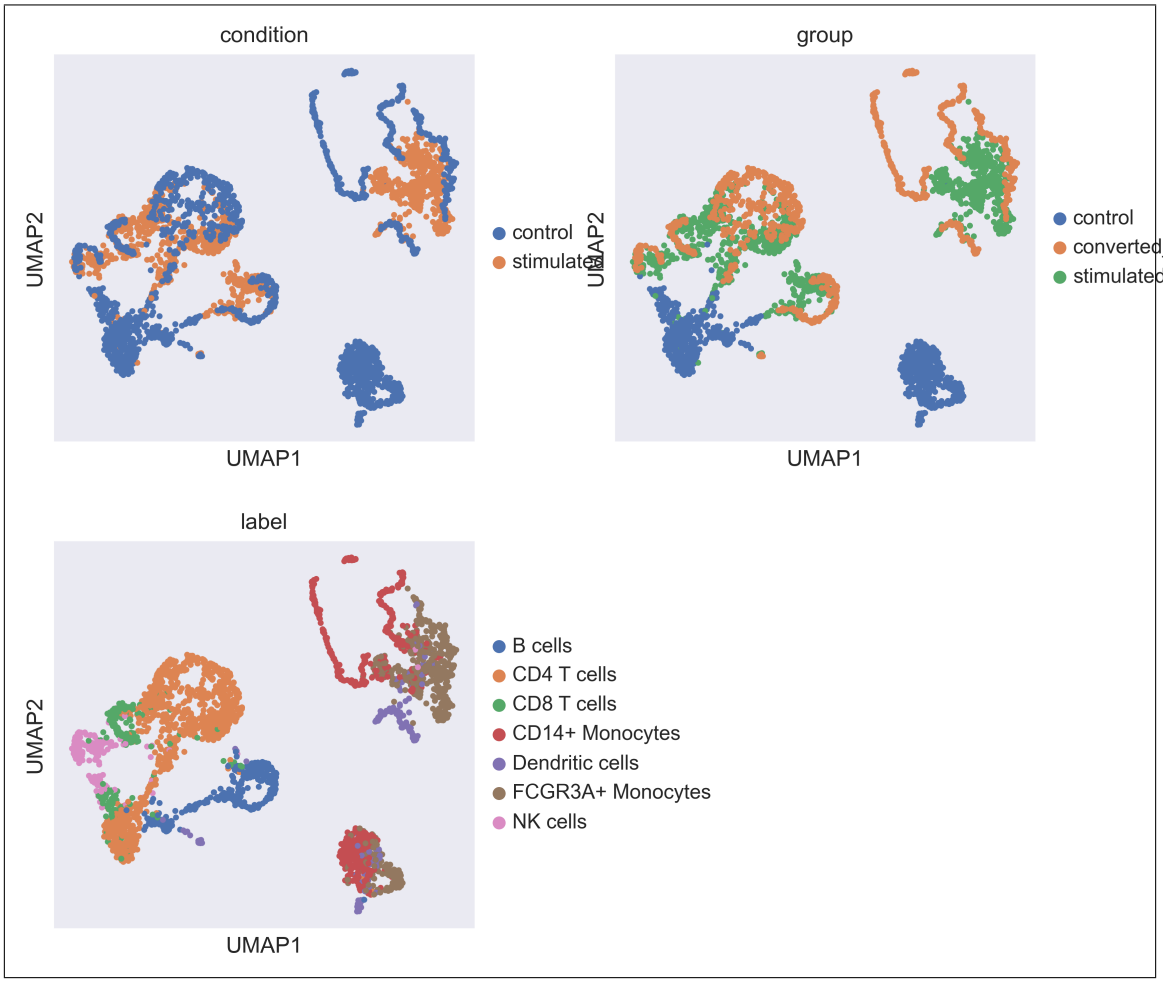
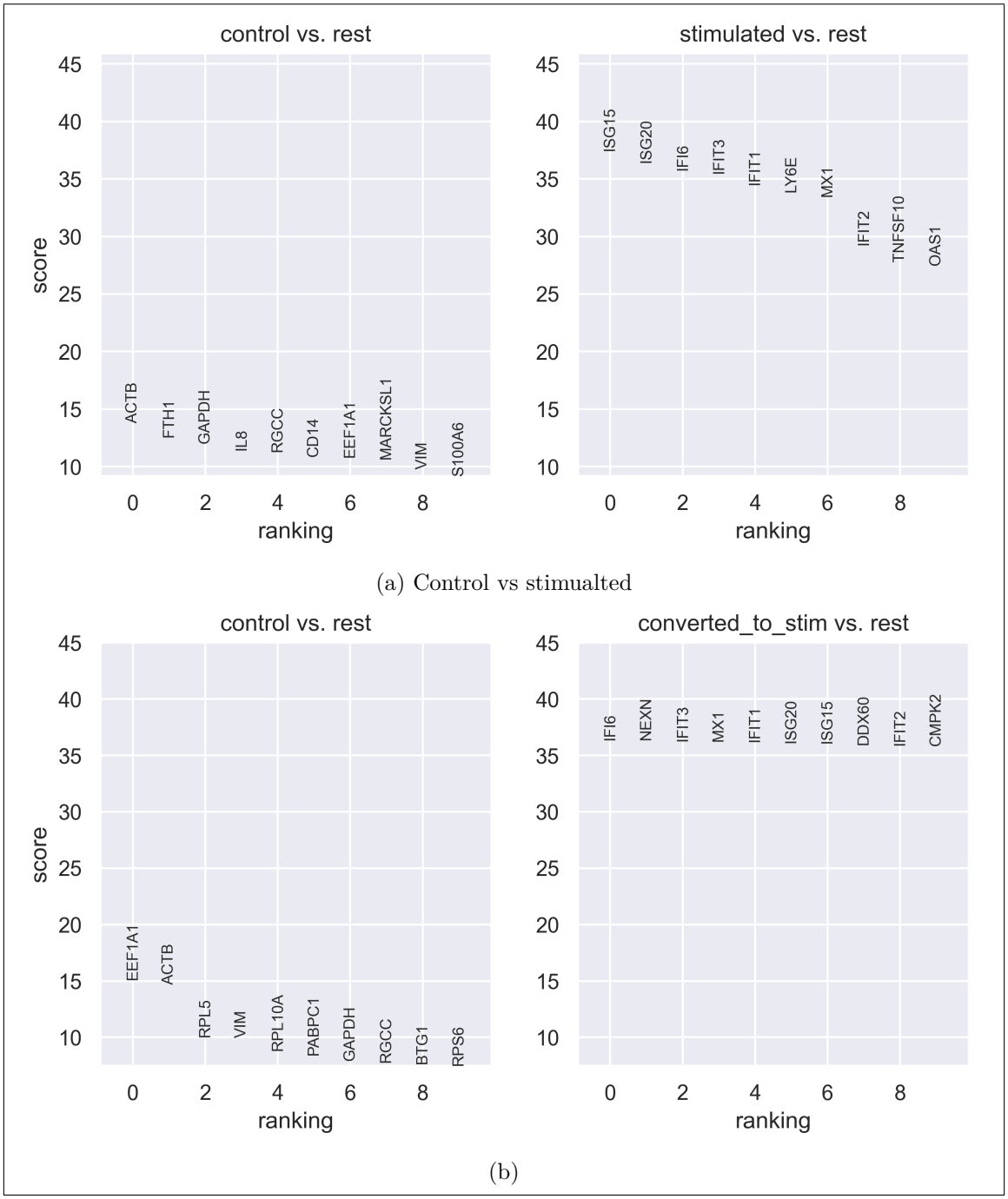


Figure 10.10: UMAP of the combined validation subset with its control group remapped into stimulated state.

Figure 10.10 shows a UMAP of the x (input/reconstructed) space of the validation subset combined with the control subset remapped into stimulated state by the c∗GMΔVÆ . We can see that the decoder learned to preserve batch effect in the x space just by using the condition information, since in the latent w space the batch effect was deliberately erased. Moreover, we see that cells of different types map close to each other, so for example the B-cells from the control group are mapped (after condition flipping) to B cells of the stimulated group. The mapping though appear somewhat one dimensional. Especially the CD14 cells end up in a long ”chain”. If we used sampling instead of taking means we would end up with a graph that has more volume to it but we chose to take the mean as

they are the best approximation that the model can give us.

Gene ranking was performed to see if the genes that are good markers for control vs stimulated are also good at marking control vs the cells converted to stimulated state. As figure 10.11 shows, among the top 10 marker genes in the two cases, there are many that are common.



(a) Control vs stimualted

(b)

Figure 10.11: control vs. control–converted–to–stimulated

A single cell class was picked (B cells) and regression comparison was performed, using the same method (in fact the same function) as in [14]. This method comparing mean expression of genes between 2 groups (again only for one cell type). As figs. 10.12 to 10.14 show, Stimulated and converted–to–stimulated have a very close to perfect correlation correlation, whereas comparing control vs. either one of them yields equally ”bad”

regressions.

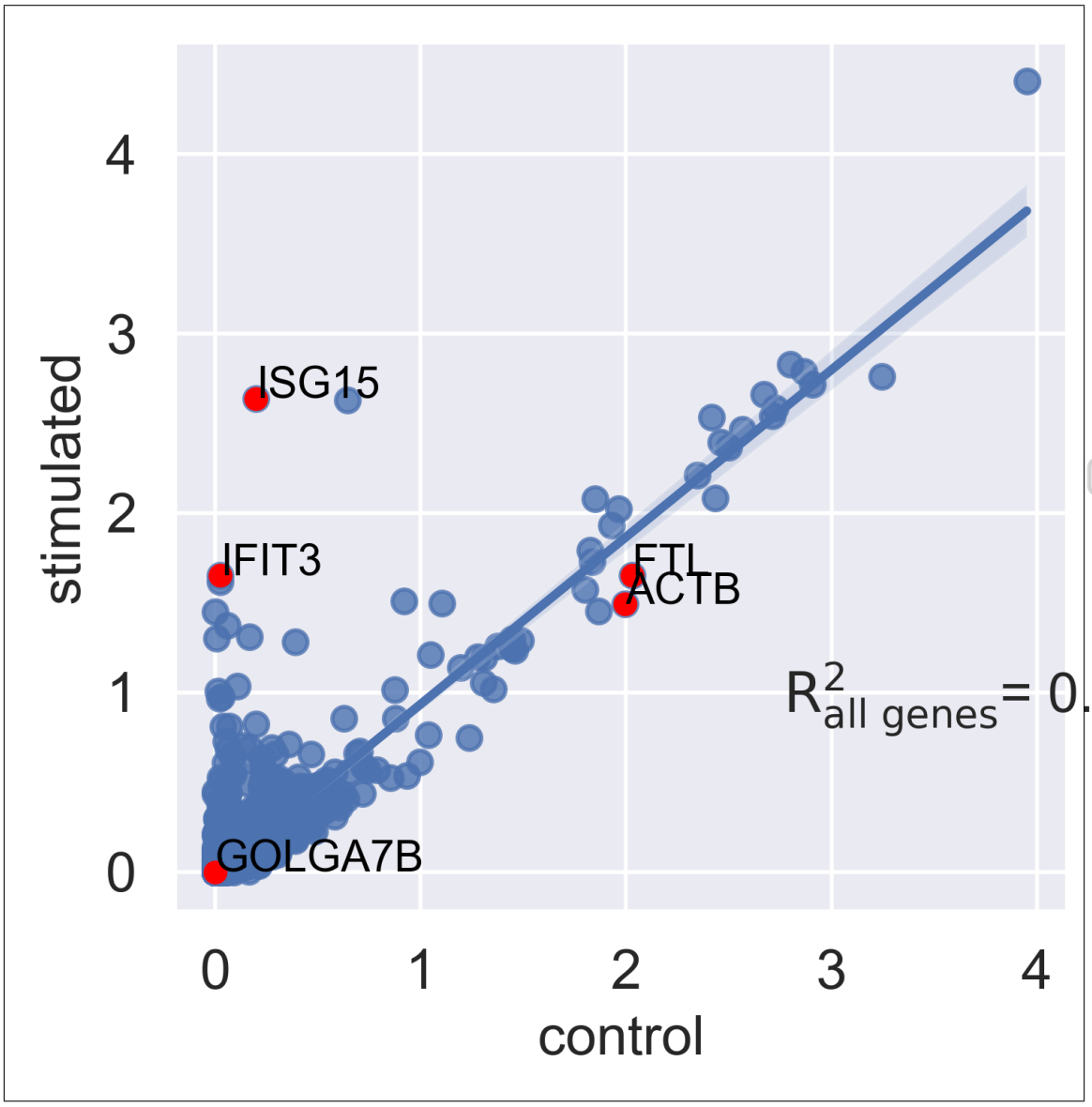


Figure 10.12: B Cells mean expression regression, control vs stimulated.

The same regression of mean expression comparisons was tested, but this time over all the cells rather than on just one cell type. The results are a repeat of the B-cells case.

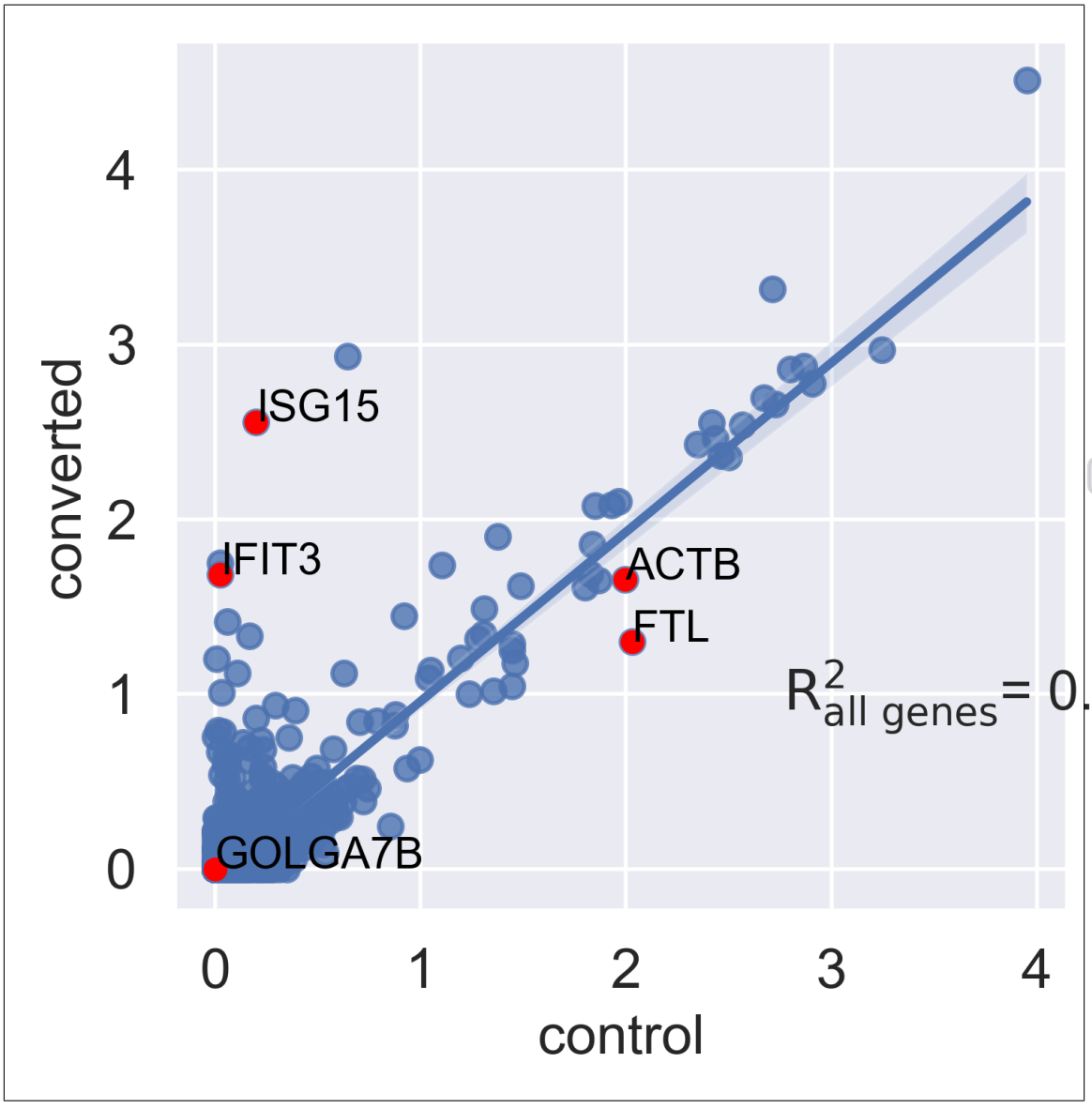


Figure 10.13: B Cells mean expression regression, control vs control–converted–to–stimulated.

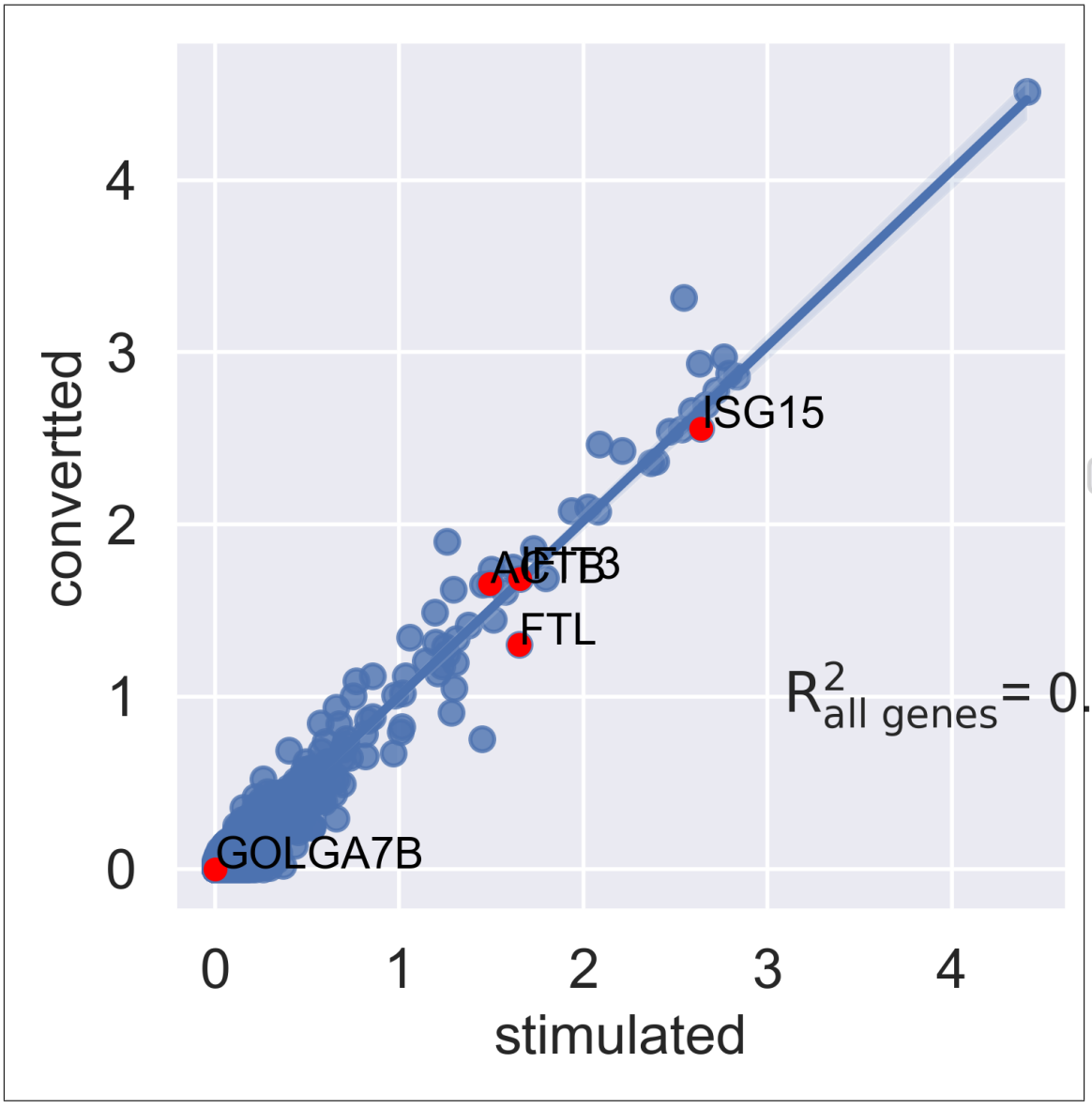


Figure 10.14: B Cells mean expression regression, stimulated vs control–converted–to–stimulated.

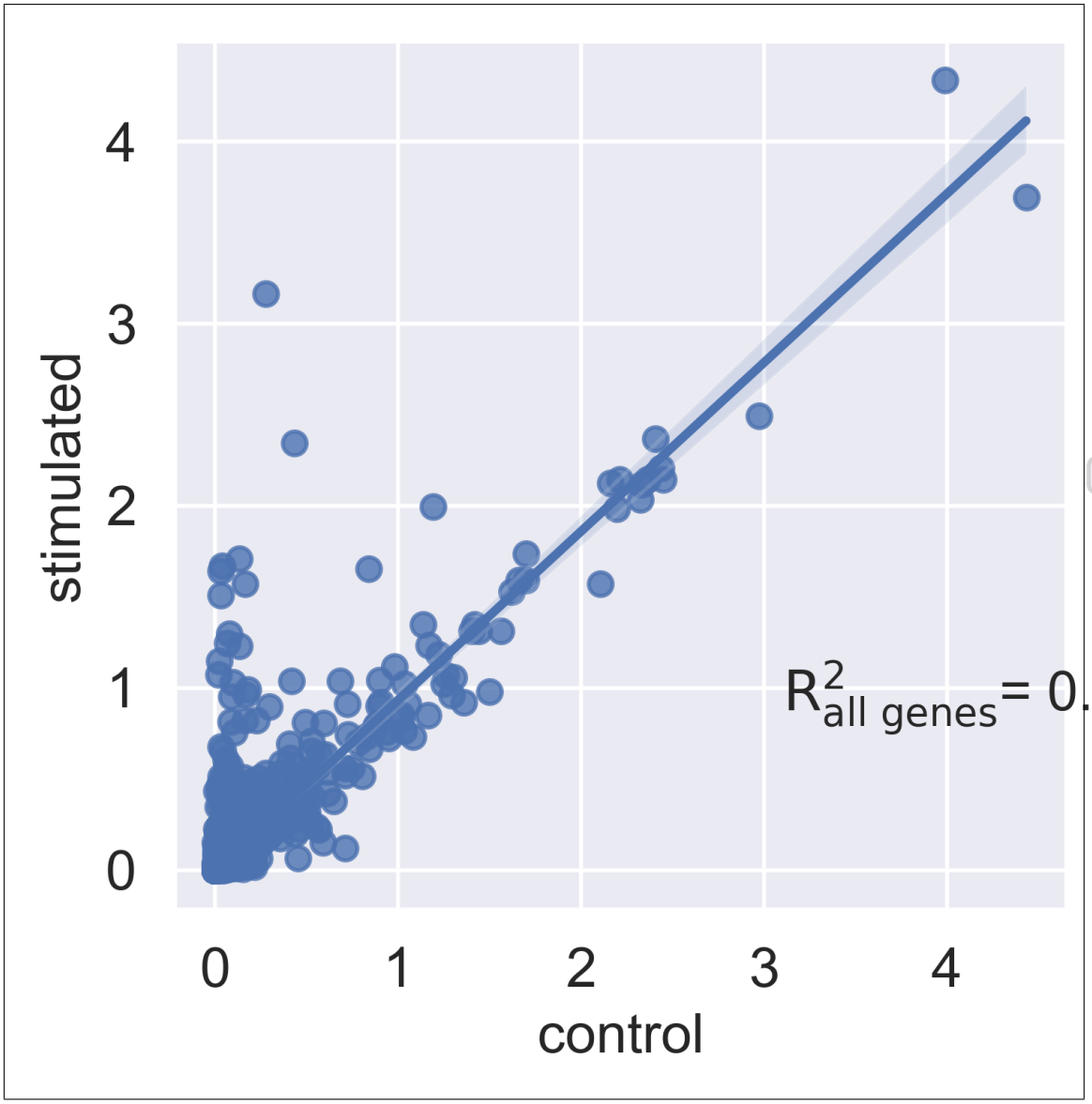


Figure 10.15: All cells mean expression regression, control vs stimulated.

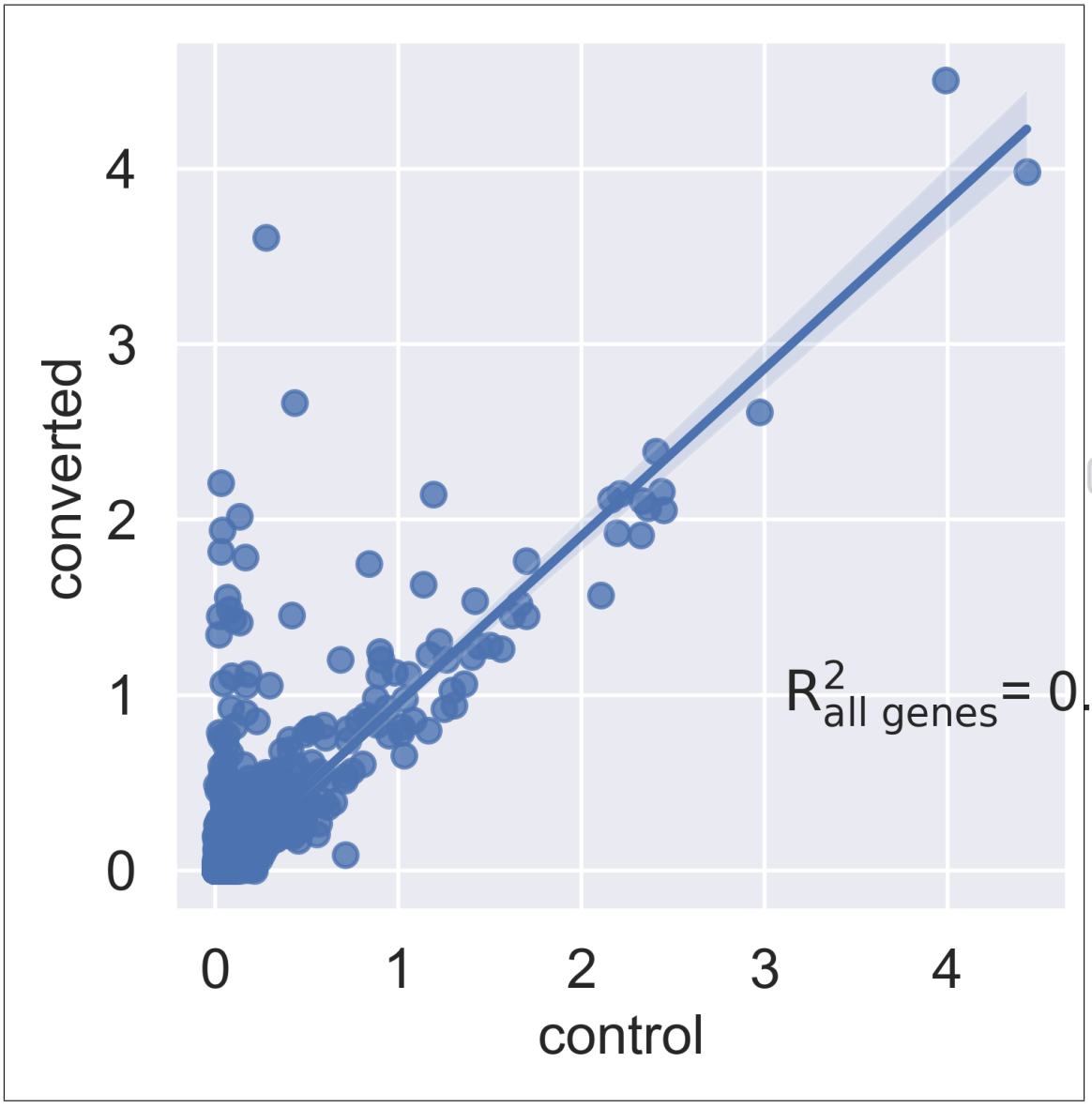


Figure 10.16: All Cells mean expression regression, control vs control–converted–to–stimulated.

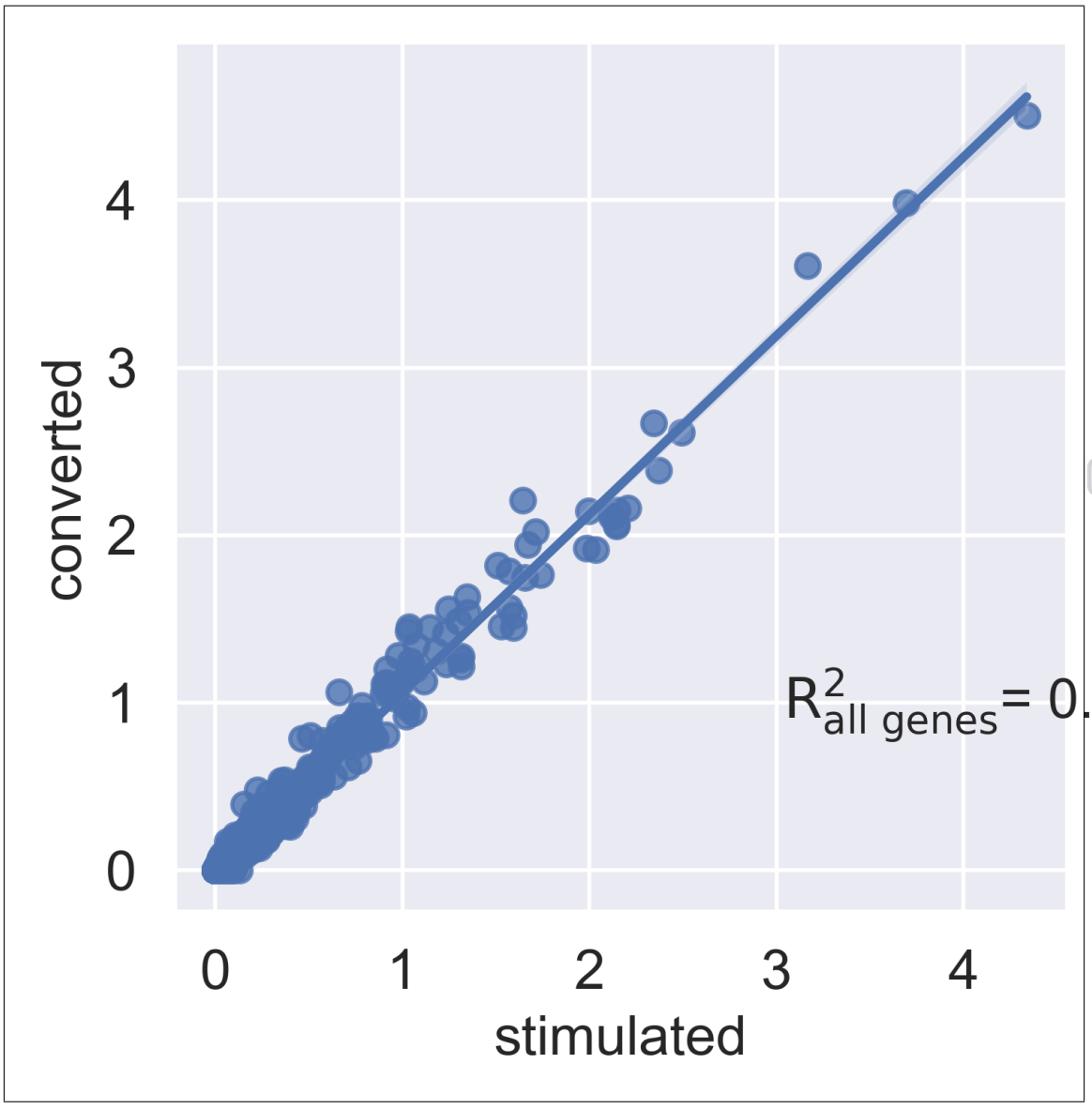


Figure 10.17: All Cells mean expression regression, stimulated vs control–converted–to–stimulated.

10.4 Generating data

What was done in the previous section didn't require us to use c∗GMΔVÆ because we kept the cell type information known anyone. A conditional VAE might produce similar results. In this section it was attempted to generate data that looks like the original Kang dataset. In order to generate cells of specific type "on demand" that we need the mixture model of the c∗GMΔVÆ .

The process goes as follows:

- sample from the w space using standard normal distribution.
- use the decoder to map each sampled w into all the mixture components on the z space, and do that for all conditions.
- Map each of the sampled z points into the reconstruction x space, using the decoder.

Since we have 2 conditions and 7 classes in this case, each sample w is used to generate 14 different samples in the latent z and then x space.

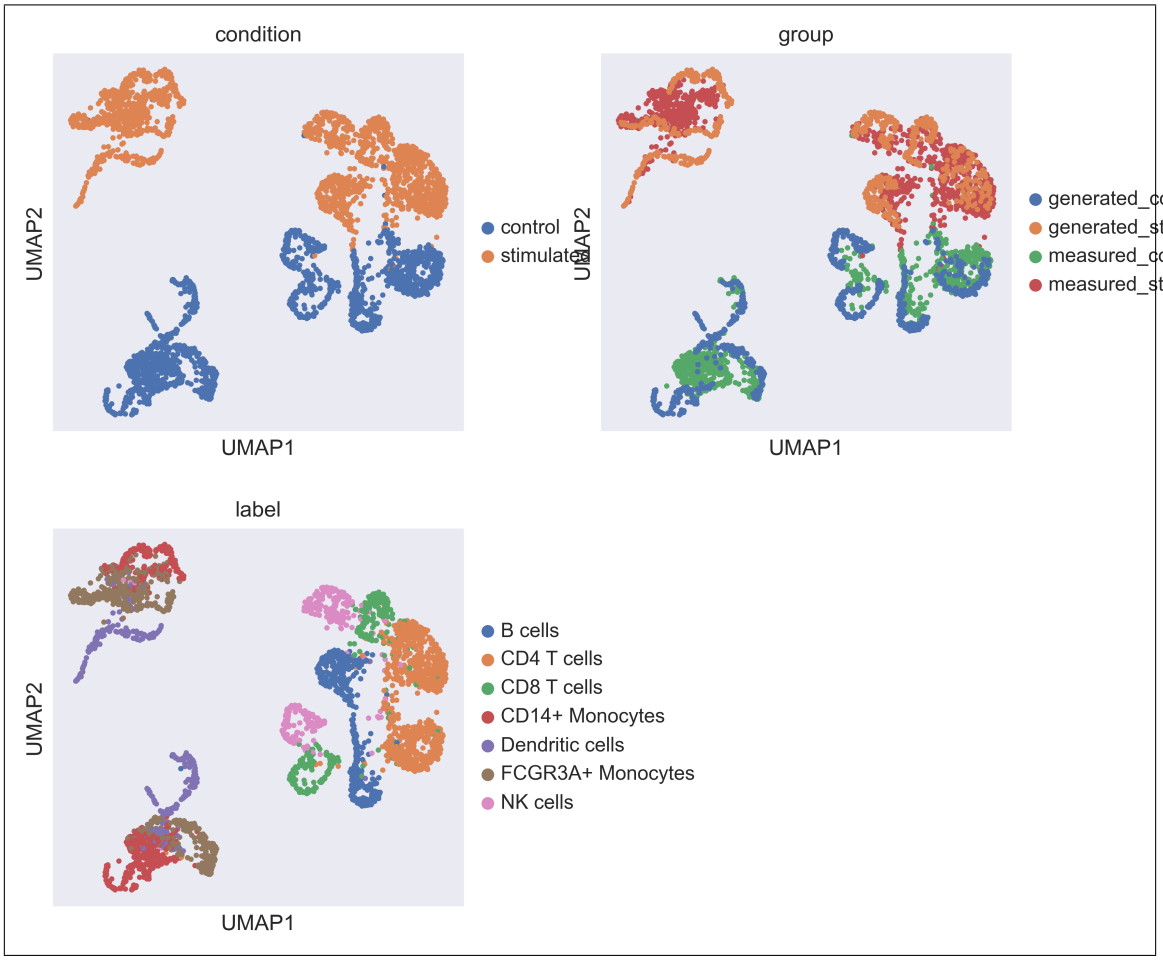


Figure 10.18: The Kang UMAP of real data combined with randomly generated data.

Figure 10.18 and 10.19 show the final result presented for this paper. It is a UMAP of a hybrid dataset, comprised from the Kang validation dataset and randomly generated samples by c∗GMΔVÆ . We can see that "synthetic" stimulated (control) data appears

next to real stimulated (control) data. So the decoder preserves batch effect also for random samples from the latent space. The class labels of the real and the generated data also appear together or nearby.

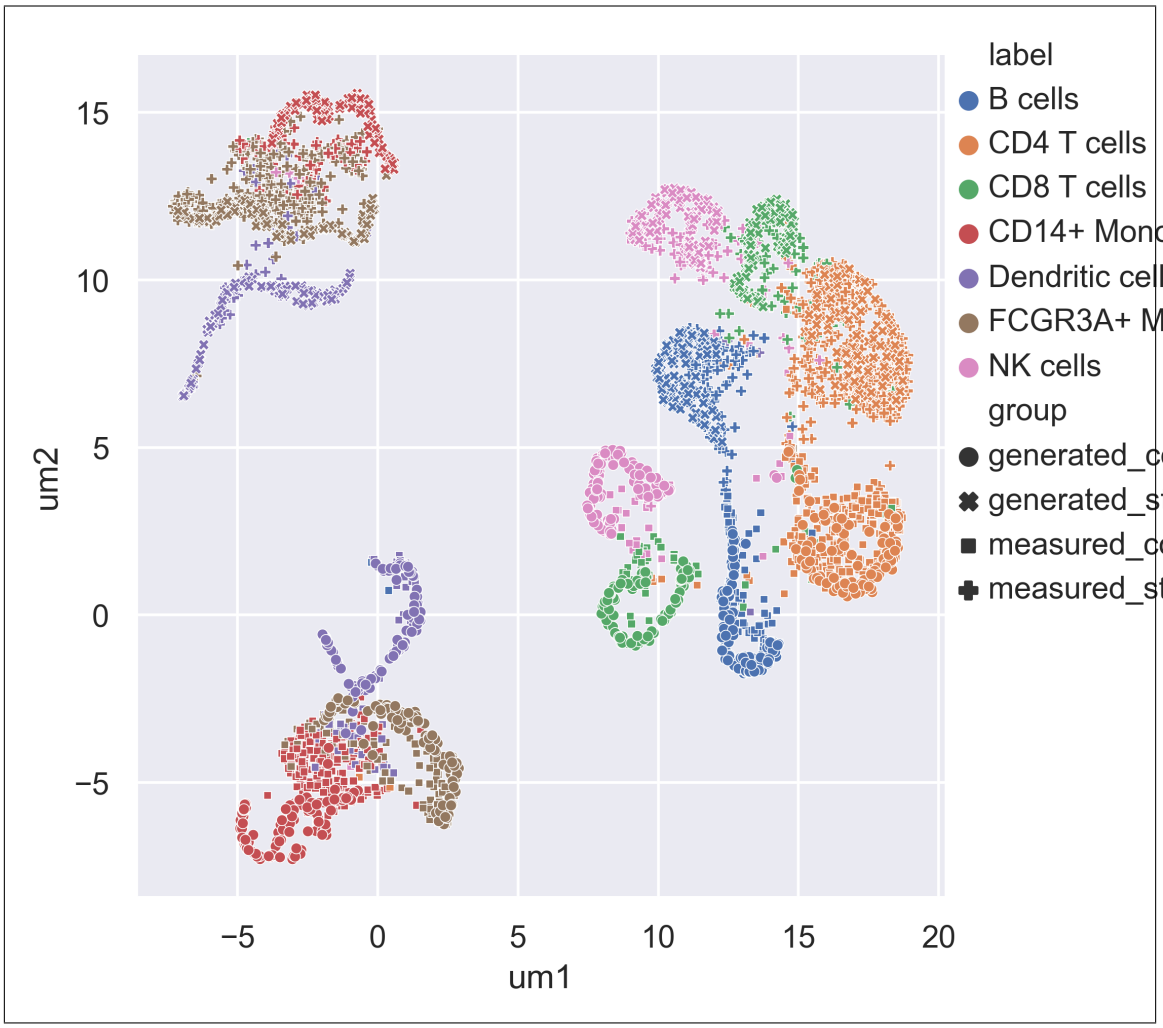


Figure 10.19: The Kang UMAP of real data combined with randomly generated data. Generated data appears close together with real data of the same label and condition.

Chapter 11

Discussion, some remarks and conclusions

The tests with synthetic conditional-mixture dataset 9 suggest that c∗GMΔVÆ is a correct VAE model for such distribution. With ”real world” data, it gets more difficult. Is the model too ”weak” or is the underlying distribution of the real data not sufficiently conditional-mixture (or the mixture component not sufficiently Gaussian)?

To make the model ”stronger” its neural network architecture needs to be augmented. It wouldn't do to just add more fully connected layers, this approach doesn't scale well. Using proven convolution networks (Resnets and the likes) should improve performance on image datasets but probably not so as much on scRNASeq data where the gene order is arbitrary unlike in images. There may be better suited architectures for scRNAseq or a need to invent such.

Changing the distribution model is another thing that may be worth trying. I am thinking of a distribution type that is ”transition–mixture”, meaning that the mixture component are not fixed with sharp switches but rather there is gradual transition and samples have compound states rather than belonging to one pure mixture component. That is just an amorphic idea at this stage, not concrete implementation.

As is stated in the abstract, the model performed quite well in the (semi)supervised case even with complex and noisy dataset. I don't know if there is demand for data generation in the realm of scRNAseq but in other types of data there is. There is an interesting use case called VADEERS [13] which is a ”drug recommender system” where the GMVAE is a subsystem of a larger neural network.

In the last chapter, the section about simulating treatment effects is based on similar things that are shown with scGen [14] using their mean-expression regression plot function on one of the datasets they used. Unfortunately due to time troubles I couldn't present an apples to apples comparison with scGen. In that paper, they show that scGen beats a conditional VAE which they compared with scGen. I am quite confident c∗GMΔVÆ will perform better.

The code written for this project and all software tools which were used are free and open source (FOSS). There is a tutorial Jupyter-lab which demonstrates how to use the code. It can be found here [11]. The entire material used in this thesis can be found here [12].

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] *Fashion-MNIST, a drop-in replacement for MNIST*. URL: `https://github.com/zalandoresearch/fashion-mnist`.

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

[5] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[6] 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.

[7] Hyun Min Kang et al. "Multiplexed droplet single-cell RNA-sequencing using natural genetic variation". In: *Nature biotechnology* 36.1 (2018), pp. 89–94.

[8] Diederik P Kingma, Max Welling, et al. "An introduction to variational autoencoders". In: *Foundations and Trends® in Machine Learning* 12.4 (2019), pp. 307–392.

[9] Diederik P Kingma and Max Welling. "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114* (2013).

[10] Durk P Kingma et al. "Semi-supervised learning with deep generative models". In: *Advances in neural information processing systems* 27 (2014).

[11] Yiftach Kolb. *The "official" c*GMΔVAE project git*. URL: `https://github.com/zelhar/mg22`.

[12] Yiftach Kolb. *The Github project housing this thesis*. URL: `https://github.com/zelhar/mg22`.

[13] Krzysztof Koras et al. "A generative recommender system with GMM prior for cancer drug generation and sensitivity prediction". In: *Machine Learning in Computational Biology*. PMLR. 2022, pp. 61–73.

[14] Mohammad Lotfollahi, F Alexander Wolf, and Fabian J Theis. "scGen predicts single-cell perturbation responses". In: *Nature methods* 16.8 (2019), pp. 715–721.

[15] Malte D Luecken and Fabian J Theis. "Current best practices in single-cell RNA-seq analysis: a tutorial". In: *Molecular systems biology* 15.6 (2019), e8746.

[16] Pablo Sánchez Martín. "Unsupervised Deep Learning". In: ().

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

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

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

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

[21] Xinyu Que et al. "Scalable community detection with the louvain algorithm". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2015, pp. 28–37.

[22] Oleh Rybkin, Kostas Daniilidis, and Sergey Levine. "Simple and effective VAE training with calibrated decoders". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 9179–9189.

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

[24] *The MNIST Database*. URL: `http://yann.lecun.com/exdb/mnist/`.

[25] F Alexander Wolf, Philipp Angerer, and Fabian J Theis. "SCANPY: large-scale single-cell gene expression data analysis". In: *Genome biology* 19.1 (2018), pp. 1–5.

[26] Grace XY Zheng et al. "Massively parallel digital transcriptional profiling of single cells". In: *Nature communications* 8.1 (2017), pp. 1–12.