List of topics to cover

With section titles and brief explantions.

Yiftach Kolb

Berlin, October 24, 2022

Abstract

punkt. punkt.

Declaration

punkt. punkt.

Acknowledgement

punkt.[5] punkt. bip/bop/boop **XxZzYyWw**$\mathbb{Z}$ so-so–so—so

List of Tables

List of Figures

Contents

Chapter 1

Introduction

punkt. *punkt*, *punkt*.

Chapter 2

Notations and definitions, preliminary concepts

2.1 Basic notations

2.1.1 Vectors, matrices and tensors

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

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

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

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

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

$(\mathbf{X}, \mathbf{Y})$ represents concatenation of two matrices which implicitly means they have the same number of rows.

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

Definition 2.1. 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

$$\|X\|_F^2 \triangleq \operatorname{trace}(\mathbf{X}\mathbf{X}^*) = \sum_{i=1}^{m} \|\mathbf{x}_i\|_2^2 = \sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}^2 \tag{2.1}$$

2.1.2 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 means that implicitly has additional variables that 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 parameter set which we don't always specify its domain and we may not always subscript ϕ with it.

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 as linear map with one extra variable which is held fixed $x_0 \equiv 1$: $f(x_0, \dots x_n) = b + a_1 x_1 + \dots a_n x_n$. b is called the *bias* of the map.

2.2 The data

we assume that the input data unless otherwise stated is real-valued matrix. Rows represent *samples* and columns represent *variables*. We assume that each raw is a realisation 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 who was realized by said observation.

We deal with two type of concrete datasets in this thesis. One of them is Single cell RNAseq data. This 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 type 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 indicate which digit it shows. Since there are 10 classes of 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 correspoinding image.

Definition 2.2. A *data matrix* is a real–valued matrix $\boldsymbol{X} \in \mathbb{R}^{N \times n}$ which represent a set of N n-dimensional data points. The N rows are also called *observations* and the n columns are *variables*.

Definition 2.3. A *class matrix*, or also a *condition matrix* $\boldsymbol{C} \in \mathbb{R}^{\mathbb{N}\times c}$ is simply 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 $(\forall k \in 1,\ldots,c)\boldsymbol{C}[i,k] = \delta_{jk}$.

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 it is a distribution matrix, namely 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.

2.2.1 Input set and target set

Sometimes the data paired into the input data $\mathbf{X}$ and the target data $\mathbf{Y}$, representing for example, samples from some unknown function $f(\mathbf{x}) = \mathbf{y}$ that we want to "learn" to represent. In classification tasks for example, $\mathbf{X}$ can be for example a set of images, and $\mathbf{Y}$ can be their labels.

In the case of autoencoders, the target set is also the input set and f is the identity (in this case f is known but we want to learn an efficient way to represent the data).

When our data comes in several matrices, for example it could be a single cell RNAseq data with normalized gene expression data $\mathbf{X}$, and cell types as target matrix $\mathbf{Y}$. It means they have must the same number of rows. And when we speak about paired input/target $\mathbf{x}, \mathbf{y}$ it means some $(\mathbf{x}, \mathbf{y}) \in (\mathbf{X}, \mathbf{Y})$ so it should be clear they belong to the same sample (row).

2.3 Linear algebra preliminary: SVD and PCA

In the following 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 [10].

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

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

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

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

(column) right eigenvectors of $\mathbf{X}^*\mathbf{X}$, sorted by the eigenvalues, where $\boldsymbol{V}_r = (\boldsymbol{v}_1|\dots|\boldsymbol{v}_r) \in \mathbb{R}^{n\times r}$ 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

$$\mathbf{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^* \tag{2.2}$$

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 $l \le r$, 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

$$\boldsymbol{V}_l = \text{argmin}_{\mathbf{W}}\{\|\mathbf{X} - \mathbf{X}\boldsymbol{W}\boldsymbol{W}^T\|_F^2 \quad : \quad \boldsymbol{W} \in \mathbb{R}^{n\times l}, \boldsymbol{W}^T\boldsymbol{W} = \boldsymbol{I}_l\} \tag{2.3}$$

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

If we consider the more general minimization problems:

$$\min_{\boldsymbol{E},\boldsymbol{D}}\{\|\mathbf{X} - \mathbf{X}\boldsymbol{E}\boldsymbol{D}\|_F^2 \quad : \quad \boldsymbol{E}, \boldsymbol{D^T} \in \mathbb{R}^{n\times l}, \} \tag{2.4}$$

$$\min_{\mathbf{W}}\{\|\mathbf{X} - \mathbf{X}\boldsymbol{W}\boldsymbol{W}^\dagger\|_F^2 \quad : \quad \boldsymbol{W} \in \mathbb{R}^{n\times l}, \} \tag{2.5}$$

It can be shown [8] that the last two problems 2.4, 2.5 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 2.4 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).

2.4 Neural networks

We briefly discuss here some of the basics of neural network to provide clarity and motivation. Mostly based on [7].

2.4.1 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 $f(a, b, x)$.

In general we can define one or more multivariate functions $g : \mathbb{R}^{n+m} \to \mathbb{R}^k$ (for simplicity of the discussion lets assume it is defined everywhere) and partition the set of its variables into 2. $g_{\mathbf{w}}(x) \triangleq g(\mathbf{w}, \mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^n, \mathbf{w} \in \mathbb{R}^m$ and let $g_{\mathbf{w}}(\mathbf{x}) := g(w, x) \in \mathbb{R}^k$.

We call a class $\mathcal{F}$ of parameterized functions *universal* if every continuous function can be uniformly approximated (inside a bounded domain) by functions 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
- derivable 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).

However these requirements are still 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. For 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 $\binom{m+n}{n}$ monomial terms.

We want this class of simpler functions, that are almost as simple as linear, and yet that suits well for statistical learning. For example we want 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".

2.4.2 Neurons

So what a "neuron"?. 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 2.1.2) 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 $x_i x_j \ldots$.

Definition 2.4. 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).

If $\mathbf{x} = (x_1, \ldots x_n) \in \mathbb{R}^n$ then $\sigma(\mathbf{x})$ is the element-wise application $\sigma(\mathbf{x}) \triangleq (\sigma(x_1), \ldots, \sigma(x_n))$.

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 2.5. Let $\sigma : \mathbb{R} - > \mathbb{R}$ be an activation function and let $f_{\mathbf{w}} : \mathbb{R}^n - > \mathbb{R}$ 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.

Think of the weights of a neuron as some mutable, tunable property, some sort of memory.

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

Connecting many neurons together can create powerful parameterized functions which we call neural networks. Connecting means that the output of one neuron is the input to one of the variables of a different neuron. 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 2.6. 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 we add more neurons and add depth to the

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

The label inside the neuron describes its activation function. In the diagrams, we let σ represent the sigmoid function. We represent the identity function either by the name of the variable (x_1, y etc.) it acts on or simple by *id*. We let the label 1 represent the constant function. We need the constant function because with it we can represent any affine map as a linear map with the first input always clamped to 1. But connecting 1 to every (non-input level) neuron would clutter the graph so it is not shown in most diagrams but still implicitly assumed. A directed edge between neurons means that the output of the neuron at the tail is multiplied by the edge weight and assigned to the input variable of the neuron it connects to. 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.

A Node's output is therefore only dependent on the output of its direct ancestral nodes (plus the bias which is usually not shown).

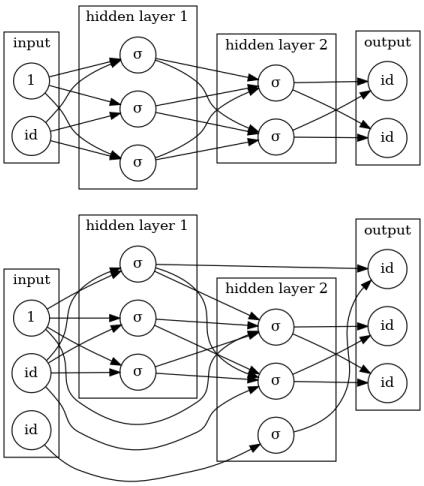


Figure 2.2: The network in the top didn't use rule 5 2.6 in the construction. It is strictly hierarchical and there are only edges between nodes of two consecutive layers. The one on the bottom is more general.

It turns out [7] 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

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.

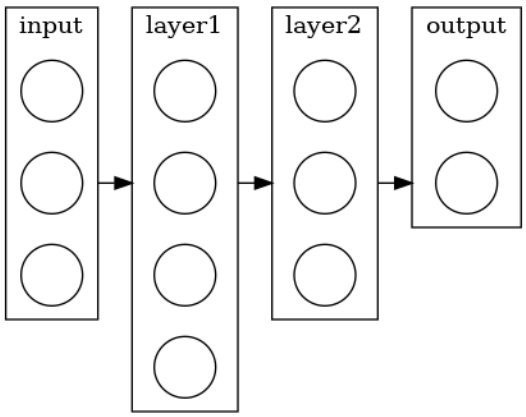


Figure 2.3: A graph of a hierarchical feed forward neural network where the connections are abstracted. Edges between layers may represent in this case a fully connected layer (every neuron has incoming edges from all neurons of the previous layer) but it could also be used for describing a convolution.

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.

2.4.3 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 2.7. Let $\phi : \mathbb{R}^n \to \mathbb{R}^m$ be a neural network. A *loss function* is a differentiable function $\mathcal{L} : \mathbb{R}^{m+m} \to \mathbb{R}$. 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 2.8. 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 bath. 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 2.9. Let ϕ be a neural network as defined in 2.6 and let $\mathcal{L}$ its associated loss function as defined in 2.7—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:

$$\phi(\boldsymbol{x}) \triangleq \{\phi(\boldsymbol{x}_i)\}_{i=1}^{m} \in \mathbb{R}^{b \times m} \tag{2.6}$$

$$\mathcal{L}(\phi(\mathbf{x}), \mathbf{y}) \triangleq \frac{1}{b} \sum_{i=1}^{b} \mathcal{L}(\phi(\mathbf{x}_i), \mathbf{y}_i) \in \mathbb{R} \tag{2.7}$$

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 sampled from the unknown data distribution $P(\mathbf{x}, \mathbf{y})$. Then equation 2.7 can be reformulated as the expected loss [1]:

$$\mathcal{L}(\phi(\mathbf{x}), \mathbf{y}) \approx \mathrm{E}_{\mathbf{x},\mathbf{y} \sim P(\mathbf{x},\mathbf{y})} \mathcal{L}(\phi(\mathbf{x}), \mathbf{y}) \tag{2.8}$$

2.4.4 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 2.10. 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:

$$\min_{\omega} \mathcal{L}(\phi_\omega(\mathbf{X}), \mathbf{Y}) \tag{2.9}$$

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 [7]. 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 [3], 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 [9]) and can be regarded as a black box.

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 $\mathbf{Y} = \mathbf{X}$.

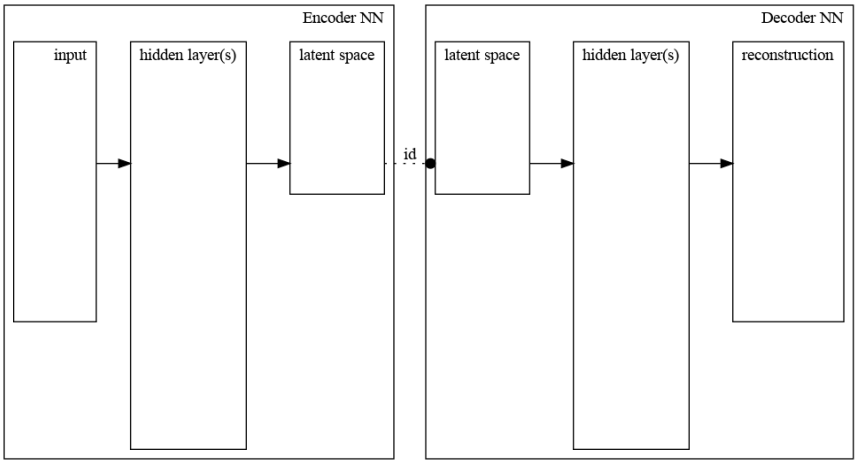


Figure 2.4: A graphic description of a "vanilla" autoencoder.

Definition 2.11. 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.

2.5.1 Relation between PCA and AE

For **centered** data, meaning 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 2.3; Whereas a **linear** autoencoder solves equation 2.5. As mentioned, it must hold that $E = D^\dagger$ (the encoder must be the Moore-Penrose inverse of the decoder).

A linear autoencoder (an AE where ϕ is linear) is therefore almost equivalent to PCA [8], in that in 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.

Chapter 3

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

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 thought of a realization of a random vector, which we also denote as $\mathbf{x}_i$ (as explained in the notation section) and similarly $\mathbf{X}$ represent both the set of r.vs as well as the realization itself. We assume that the $\boldsymbol{x}_i$'s are independent and identically distributes (i.i.d) random vectors with some distribution function $p(\mathbf{x})$ 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 $p(\boldsymbol{x})$. Then $\log p(\boldsymbol{X}) = \sum_1^N \log p(\boldsymbol{x}_i)$ is called the *log evidence* of our data.

The r.vs $\boldsymbol{X}$ are high dimensional however we have some reason to believe that behind the scenes there are some hidden (latent), smaller dimensional, r.vs $\boldsymbol{Z} = \{\boldsymbol{z}_1 \dots \boldsymbol{z}_N\}$ that generate the observations $\boldsymbol{X}$. In other words we think that $\boldsymbol{X}$ is conditioned on $\boldsymbol{Z}$ and we can speak of the joint distribution $p(\boldsymbol{X}, \boldsymbol{Z}) = p(\boldsymbol{X}|\boldsymbol{Z})p(\boldsymbol{Z})$. Because we assume i.i.d for both $\boldsymbol{X}$ and $\boldsymbol{Z}$ all the distributions factor over the individual samples multiplicatively, e.g. $p(\boldsymbol{X}|\boldsymbol{Z}) = \prod p(\boldsymbol{x}_i|\boldsymbol{z}_i)$, $p(\boldsymbol{X}) = \prod p(\boldsymbol{x}_i)$ and so forth. This makes working with log probabilities very easy.

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" $\boldsymbol{Z} = (\boldsymbol{z}_1, \boldsymbol{z}_2, \dots)$, where each $\boldsymbol{z}_i$ is some multidimensional r.v and possibly composed of several simpler r.vs (for example a categorical and a normal r.vs). 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 a 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. The *evidence lower bound (ELBO)* with respect to p, q is:

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

$$-\mathcal{L}(q, p) \triangleq -\mathcal{L}(q, p, \mathbf{X}) = \int \log \frac{p(\boldsymbol{X}, \boldsymbol{Z})}{q(\boldsymbol{Z})} dq(\boldsymbol{Z}) \tag{3.2}$$

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 *log evidence*. (using Jensen's inequality)

$$\begin{aligned} \log p(\boldsymbol{X}) &= \log \int p(\boldsymbol{X}, \boldsymbol{Z}) d\boldsymbol{Z} = \log \int \frac{p(\boldsymbol{X}, \boldsymbol{Z})}{q(\boldsymbol{Z})} q(\boldsymbol{Z}) d\boldsymbol{Z} \\ &= \log \int \frac{p(\boldsymbol{X}, \boldsymbol{Z})}{q(\boldsymbol{Z})} dq(\boldsymbol{Z}) \geq \int \log \frac{p(\boldsymbol{X}, \boldsymbol{Z})}{q(\boldsymbol{Z})} dq(\boldsymbol{Z}) \triangleq -\mathcal{L}(q, p, X) = -\mathcal{L}(q, p) \end{aligned} \tag{3.3}$$

In equation 3.3 we found a lower bound $\mathcal{L}(q, p)$ for the log evidence $\log p(\boldsymbol{X})$, 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 log evidence and it is equivalent to minimizing $KL(q(\boldsymbol{Z}) \| p(\boldsymbol{Z}|\boldsymbol{X}))$:

$$\begin{aligned} -\mathcal{L}(q, p) &\triangleq \int \log \frac{p(\boldsymbol{X}, \boldsymbol{Z})}{q(\boldsymbol{Z})} dq(\boldsymbol{Z}) = \int \log \frac{p(\boldsymbol{Z}|\boldsymbol{X}) p(\boldsymbol{X})}{q(\boldsymbol{Z})} dq(\boldsymbol{Z}) \\ &= \int \log p(\boldsymbol{X}) dq(\boldsymbol{Z}) - \int \log \frac{q(\boldsymbol{Z})}{p(\boldsymbol{Z}|\boldsymbol{X})} dq(\boldsymbol{Z}) = \log p(\boldsymbol{X}) - KL(q(\boldsymbol{Z}) \| p(\boldsymbol{Z}|\boldsymbol{X}) \end{aligned} \tag{3.4}$$

We can rewrite equation 3.4 as:

$$\log p(\boldsymbol{X}) = -\mathcal{L}(q, p) - KL(q(\boldsymbol{Z}) \| p(\boldsymbol{Z}|\boldsymbol{X})) \tag{3.5}$$

Equation 3.5 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. In this case let θ represent the set of parameters for p, and ϕ the parameters for q. Meaning we are dealing with a family of distributions $p_\theta(x, z)$ and another family $q_\phi(z)$.

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

We assume that we can only approach the "real" distribution using θ from below $\log p(X) \geq \log p_\theta(X)$. So together with equation 3.3 we have

$$(\forall\theta, \phi) \log p(X) \geq \log p_\theta(X) \geq -\mathcal{L}(q_\phi) = \int \frac{p_\theta(X, Z)}{q_\phi(Z)} dq_\phi(Z) \tag{3.6}$$

And from equation 3.5 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 3.3 and 3.4 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 $p(\mathbf{Z})$.

$$\begin{aligned}
\mathcal{L}(q, p) &\triangleq \int -\log \frac{p(\mathbf{X}, \mathbf{Z})}{q(\mathbf{Z}|\mathbf{X})} dq(\mathbf{Z}|\mathbf{X}) = \int -\log \frac{p(\mathbf{X}|\mathbf{Z})p(\mathbf{Z})}{q(\mathbf{Z}|\mathbf{X})} dq(\mathbf{Z}|\mathbf{X}) \\
&= \int -\log p(\mathbf{X}|\mathbf{Z}) dq(\mathbf{Z}|\mathbf{X}) + \int \log \frac{q(\mathbf{Z}|\mathbf{X})}{p(\mathbf{Z})} dq(\mathbf{Z}|\mathbf{X}) \\
&= \int -\log p(\mathbf{X}|\mathbf{Z}) dq(\mathbf{Z}|\mathbf{X}) + KL(q(\mathbf{Z}|\mathbf{X})\|p(\mathbf{Z}))
\end{aligned} \tag{3.7}$$

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 3.7 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 Mean field approximation

Usually we treat the dimensions of $\mathbf{x}$, $\mathbf{z}$ etc. as independent. That means if $\mathbf{x} = (x_1, \ldots, \mathbf{x}_n)$ is a r.v. in $\mathbb{R}^n$ we assume that the x_i are independent and therefore $P(\mathbf{x}) = \prod_1^n P_i(x_i)$. Mean field approximation simplifies the implementation and speeds up the computation and has been the standard practice since the beginning of VAEs [5].

3.2.4 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 3.7.

Specifically the encoder part of the network is a feed forward neural network $f_\theta(\mathbf{z})$ which is used to define $P_\theta(x|z)$. 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 matrice. 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 $p(z)$ we set some fixed prior distribution.

Note that the the encoder network (similarly the decorer) 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 $\Theta \subseteq \mathbb{R}^k \ldots$).

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

While an autoencoder works on deterministic data, with the encoder mapping 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 $\mathbf{z}$: $\mathbf{x} \mapsto q(\mathbf{z}|\mathbf{x})$ and it maps $\mathbf{z}$ into a distribution over $\mathbf{x}$, $\mathbf{z} \mapsto p(\mathbf{x}|\mathbf{z})$.

The loss function associated with a VAE is minus ELBO. This means training the VAR maximizes equation 3.7 and therefore also the log evidence.

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

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

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

$$\begin{aligned} \mathcal{L}(p, q, \mathbf{x}) &= \int -\log \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{q(\mathbf{z}|\mathbf{x})} dq(\mathbf{z}|\mathbf{x}) \\ &= \int -\log p(\mathbf{x}|\mathbf{z}) dq(\mathbf{z}|\mathbf{x}) + \int \log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})} dq(\mathbf{z}|\mathbf{x}) \\ &\approx \frac{1}{k} \sum_{i=1}^{k} [-\log p(\mathbf{x}|\mathbf{z}_i) + \log \frac{q(\mathbf{z}_i|\mathbf{x})}{p(\mathbf{z}_i)}] \end{aligned} \tag{3.9}$$

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 [4].

3.2.6 using the decoder as data generator encoder for dimensionality reduction

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 decoded distribution $\mathbf{x} \sim D(\mathbf{z})$, in the observed space.

Given observation $\mathbf{x}$, we can non deterministically encode $\mathbf{x}$ into the latent space by taking the mean: $\mathbf{x} \mapsto \mathbf{E}[E(\mathbf{x})]$. Or we can non-deterministically draw $\mathbf{z}$ from the encoded distribution: $\mathbf{x} \mapsto \mathbf{z} \sim E(\mathbf{x})$.

3.2.7 Graphical representation

It is both convenient as well as informative to include a graphical description of our probabilistic models by way of plate diagrams.

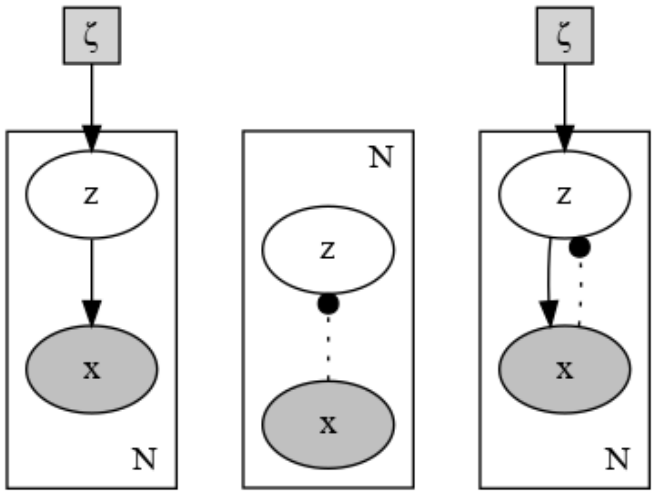
In a plate diagram nodes represent random variables and arrows represent dependency. Figure 3.1 is a plate diagram of the VAE model with slight adaptation. We use doted arrows to represent the arrows of the inference model (encoder network), and regular arrows for the generative model (decoder network). Regular (triangular) arrowhead represents real probabilistic dependency whereas rounded arrows are reminding us that this is a separate network and also to somehow mark it is not a complete probability because there is have no "prior" $q(\mathbf{x})$ to speak of.

Plate represents packing of N i.i.ds since we have N observations $X = (x_i)_1^N$ and correspondingly N latent variables $Z = (z_i)$.

Shaded node represent known values (either observation of prior).

The squared ζ node represent some fixed parameter which describes the prior distribution of $p(\mathbf{z}) := p(\mathbf{z}|\zeta)$. It is possible to make ζ a non-fixed stochastic parameter but in the case of this vanilla VAE I don't think it has any advantage and don't know of anyone who does that.

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 3.1: 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), We use mean field approximation for p and q with Gaussian distributions. We set the prior $p(\mathbf{z})$ to be diagonal standard Gaussian $p(\mathbf{z}) \sim \mathcal{N}(;\mathbf{0}, \mathbf{1})$. And $p(\mathbf{z}|\mathbf{x}) \sim \mathcal{N}(;D(\mathbf{z}))$ is a diagonal Gaussian, where the decoder determines

its means and variances $D(\mathbf{z}) = (\boldsymbol{\mu}(\mathbf{z}), \boldsymbol{\sigma}(\mathbf{z}))$, And similarly $q(\mathbf{z}|\mathbf{x}) \sim \mathcal{N}(; E(\mathbf{x}))$.

3.3 Expanding the VAE model

If we look at figure 3.1 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 $\mathbf{x}_1, \mathbf{x}_2, \dots \mathbf{x}_k$, and the set of latent random vectors and stochastic parameters $\mathbf{z}_1, \dots \mathbf{z}_l$.
2. specify how to factor the generative model $p(\mathbf{x}_1, \dots, \mathbf{x}_k|\mathbf{z}_1 \dots, \mathbf{z}_l)$
3. specify how to factor the inference model $q(\mathbf{z}_1 \dots \mathbf{z}_l|\mathbf{x}1, \dots \mathbf{x}_k)$
4. choose appropriate priors $p(z_i)$ and
5. Chose 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" as $p(\mathbf{x}, \mathbf{z}, c) = p(\mathbf{x}|\mathbf{z}, c)p(\mathbf{z}|c)p(c)$. But we can also simplify if it makes sense. Suppose we have reason to believe that $\mathbf{z}$ alone generates $\mathbf{x}$, so given $\mathbf{z}$, $\mathbf{x}$ and c are treated as independent. But $\mathbf{z}$ itself will be a mixture of k distributions. Moreover, since our data is balanced, it is clear that $p(c) = \frac{1}{k}$. The generative process is therefore as follows: draw a category $c \sim Cat(\frac{1}{k})$. Then draw $\mathbf{z} \sim p(\mathbf{z}|c)$. Then draw $\mathbf{x} \sim p(\mathbf{x}|\mathbf{z})$. So our factorization of p is: $p(\mathbf{x}, \mathbf{z}, c) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z}|c)\frac{1}{k}$.

As for the inference model, 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.

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 distribution 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.

Remember that the loss function is still the minus elbo but with our factorization it will become:

$$\begin{aligned}
\mathcal{L}(p, q, \mathbf{x}, c) &= \int -\log \frac{p(\mathbf{x}, c, \mathbf{z})}{q(\mathbf{z}|\mathbf{x}, c)} dq \\
&= \int -\log \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z}|c)p(c)}{q(\mathbf{z}|\mathbf{x}, c)} dq \\
&= \int -\log p(\mathbf{x}|\mathbf{z}) dq + \int \log \frac{q(\mathbf{z}|\mathbf{x}, c)}{p(\mathbf{z}|c)} dq + \log(k) \\
&= \int -\log p(\mathbf{x}|\mathbf{z}) dq + KL(q(\mathbf{z}|\mathbf{x}, c) \| p(\mathbf{z}|c)) + \text{const}
\end{aligned} \tag{3.10}$$

From equation 3.10, we can ignore the constant and we see again a 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 if we have for example several batches of data, for the same type of data but from different experiment. In this case we can use a CVAE model with fixed prior to reduce the batch effect.

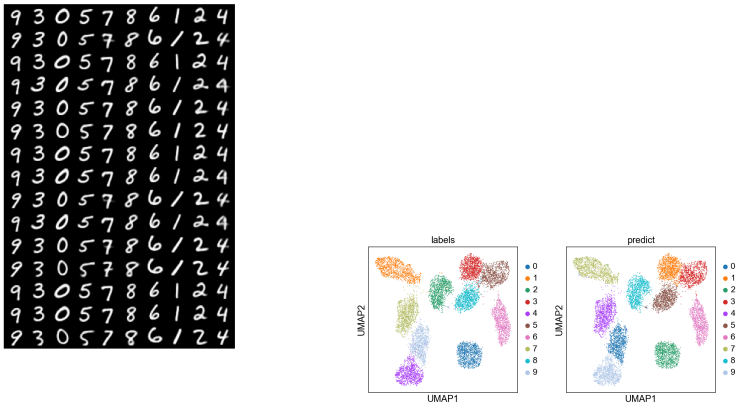
Chapter 4

Gaussian mixture model VAEs

Theoretical background and with some examples from publications and my own tests.

4.0.1 Relation between AE and VAE

4.0.2 Conditional VAE



(a) (b)

Figure 4.1: a figure

Chapter 5

Experiments and results

5.1 Tests with MNIST and FMNIST

5.2 Tests with scRNAseq Data

some words about (sc)RNAseq and published papers where AE and VAE models have been applied. What we were hoping to achieve and compare with.

Chapter 6

Discussion, some remarks and conclusions

punkt. punkt. punkt.

Bibliography

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

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

[3] 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.

[4] Diederik P Kingma and Max Welling. "An introduction to variational autoencoders". In: *arXiv preprint arXiv:1906.02691* (2019).

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

[6] Mohammad Lotfollahi, F Alexander Wolf, and Fabian J Theis. "Generative modeling and latent space arithmetics predict single-cell perturbation response across cell types, studies and species". In: *bioRxiv* (2018), p. 478503.

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

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

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

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