Momchil Peychev

Experimental Study on the Properties of Disentangled Autoencoders

Project Dissertation
Computer Science Tripos – Part II

Girton College

May 20, 2017

Proforma

Name:	**Momchil Peychev**
College:	**Girton College**
Project Title:	**Experimental Study on the Properties of Disentangled Autoencoders**
Examination:	**Computer Science Tripos, Part II, 2017**
Word Count:	**11,996**[1]
Project Originators:	**Petar Veličković and Momchil Peychev**
Supervisor:	**Petar Veličković and Dr Pietro Liò**

Original Aims of the Project

This project primarily aims to conduct a thorough investigation of disentangled autoencoders, recently proposed neural network architectures for unsupervised learning. They are reported for their ability to encode independent input generating factors in separate nodes of the neural network leading to more interpretable and predictable models. In order to do this, a generic autoencoder implementation and a flexible, easily extensible test bed have to be provided. The evaluation should be performed both on synthetic and pre-existing data sets paying particular attention to the influence of hyperparameters to the model behaviour and performance.

Work Completed

The project was a great success, meeting and extending all of the requirements. Both fully connected and convolutional disentangled autoencoders were successfully implemented and evaluated on a synthetically generated dataset. Known results from scientific literature were reproduced and new, unpublished ones were obtained. The classification ability of the autoencoders was tested against the famous benchmark dataset MNIST and it was asserted that the disentanglement acts negatively to their discriminative ability.

[1] Computed by TeXcount, `http://app.uio.no/ifi/texcount/`.

Special Difficulties

Managing the long training times of the algorithms appropriately in order to fit in the schedule. The total training time for all models was around one month.

Declaration

I, Momchil Peychev of Girton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date: May 20, 2017

Contents

List of Figures

Acknowledgements

I would like to express my sincere gratitude towards the following people who contributed with comments and suggestions during my work on this dissertation:

- **Petar Veličković**, for his considerable support and encouragement over the past eight months. His advice and guidance were invaluable and I could not wish for a more dedicated supervisor for my project.
- **Dr Pietro Liò**, for allowing me to be a part of the Computational Biology group in the Computer Lab this year and letting me use their computational resources. His comments on the presentation of my work were incredibly useful.
- My Director of Studies, **Chris Hadley**, my academic supervisors, tutors, family and friends for all of their support during my time in Cambridge.

Chapter 1

Introduction

The aim of my project was to build and thoroughly evaluate the recently published model of a disentangled autoencoder. I have successfully fulfilled my core by reproducing known results from literature, providing many extensions on top of that:

- New, unreleased experimental results were obtained;
- Apart from the originally planned fully connected architecture, a convolutional one was also implemented;
- The trade-offs between disentanglement, reconstruction and classification ability of the autoencoder were studied on a classical benchmark data set.

My inner drive when choosing the project was the desire to investigate machine learning models which provide higher level of transparency and are easier to reason about then other state-of-the-art algorithms. The rest of this chapter gives the motivations for studying this topic, an example where the proposed methods proved to be useful, and an overview of the previous work done in the field.

1.1 Motivation

The exponential growth in data availability and the rapid increase of computational power in the past decade have allowed neural network based algorithms to achieve impressive practical results in the fields of computer vision [25, 30, 34], natural language processing and generation [19, 15, 35], and game playing [32, 27] to mention a few, surpassing human performance on several complex tasks[1] [16]. Despite the undeniable potential of the "deep learning" approach, however, more research is required to better understand its limits [33, 31].

A common criticism against neural networks is their opacity. It is difficult to investigate their exact learning process and, in particular, to reason about both the learnt weights and the underlying high-level features. This is a major hindrance for deploying neural networks in safety-critical systems (e.g. self-driving cars) for the moment. After training a deep neural network, we typically cannot define an easily interpretable decision rule out of

[1] DeepMind's AlphaGo won 4-1 against Go grandmaster and world champion Lee Sedol in March, 2016: `https://en.wikipedia.org/wiki/AlphaGo_versus_Lee_Sedol`

the learnt parameters (e.g. for controlling the vehicle in the self-driving car example) and therefore, in case of an accident, it would be very hard to establish whether the problem is in malicious input data, implementation errors, or the proposed model simply not being complex enough to capture the underlying process[2].

My project primarily concerns the model of a disentangled autoencoder which represents a recent development towards ameliorating the issue, as it is capable of learning independent generating factors separately in the network, thus being more interpretable and more predictable in its behaviour.

1.2 Toy Problem

A formal definition of autoencoders will be given in the Preparation, but for the moment we can imagine them as systems, composed of two parts: an encoder, taking input data and mapping it to some internal code (typically with a smaller dimension), and a decoder which given the code, tries to reconstruct the original data. The disentangled autoencoder architecture, in particular, was originally proposed by Higgins *et al.* [17]. In order to make reproducing and extending this paper's results as transparent as possible, I have developed a toy dataset which is a slight modification to the one used for evaluating the characteristics of the model in [17]. It is therefore central to this project and will be systematically considered in greater detail throughout the Implementation and Evaluation chapters.

Assume we have a collection of images $\mathbf{X}$ where each image $\mathbf{x}_i$ has been generated by fixing particular values for each underlying generating factor, e.g. shape of a figure, position, scale, and rotation, as in Figure 1.1.

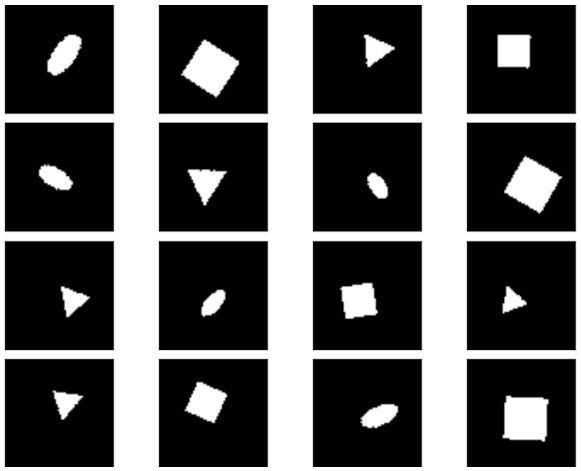


Figure 1.1: Example images generated by choosing a shape, position, scale and rotation of a figure to be visualised. These underlying features are independent of each other.

[2]Especially relevant to the two infamous Uber and Tesla crashes in less than a year:
`https://techcrunch.com/2017/03/25/uber-self-driving-test-car-involved-in-crash-in-arizona/`
`https://techcrunch.com/2016/06/30/tesla-crash/`

I will later show that under certain assumptions the disentangled autoencoder is able to automatically detect these factors and encode most of them in a *disentangled* representation. *The approach is completely unsupervised and no prior information about the number or the nature of the factors is required.* Conversely, the baseline approaches all produced highly entangled (and therefore uninterpretable) representations – small perturbations in any part of the generated code cause unintelligible changes to the decoded image.

1.3 Related Work

Autoencoders have been part of the neural network field since the late 80s [5, 21]. Because of their capability to perform dimensionality reduction, they are sometimes considered to do a more powerful non-linear Principal Component Analysis (PCA) [13, 4].

Apart from their plain variant, a denoising version was proposed by Vincent *et al.* [36]. As mentioned in the previous section, the disentangled autoencoder was initially introduced by Higgins *et al.* in [17]. It was succeeded by another paper [18], presented at ICLR[3] in April 2017, when the implementation phase of my project was mostly completed. It can be considered a generalisation of the variational autoencoder [24] devised by Kingma and Welling. Earlier attempts at disentangled factor learning either required a priori knowledge about the data generating factors [20, 29, 37, 14, 7], or did not scale well [10, 9].

Several months into the project, I discovered simple example autoencoders were implemented in TensorFlow[4] [1] and Keras[5] [8] tutorials but they were not general enough for the needs of the project. Moreover, none of them was actually an implementation of a disentangled autoencoder.

[3]International Conference on Learning Representations – `www.iclr.cc`

[4]An open-source software library for Machine Intelligence – `https://www.tensorflow.org/`

[5]Deep Learning library for Theano and TensorFlow – `https://keras.io/`

Chapter 2

Preparation

This chapter presents a theoretical overview of the major algorithms implemented in this work and then specifies the requirements that should be fulfilled by the project. Software engineering aspects necessary for its successful completion are discussed afterwards.

2.1 Theory

The aim of the following sections is to present a self-contained overview of the theory behind neural networks, and autoencoders in particular. Neural network training via backpropagation is considered in detail. In order to establish the disentangled autoencoder framework, the required theory on variational inference is presented as well.

2.1.1 Neural Networks

Neural networks are computational graphs consisted of linear transformations interspersed with nonlinearity applications. Because of this mixture of operations, if properly designed, they are capable of approximating arbitrarily complex functions.

2.1.1.1 Neural Network Architectures

A single neuron

The artificial neuron, sometimes also found in literature as a perceptron, is the building block of neural networks. It takes inputs $x_1, x_2, \ldots, x_n \in \mathbb{R}$, each of which is weighted by $w_1, w_2, \ldots, w_n \in \mathbb{R}$, indicating the relative importance of each input. The neuron computes the dot product between the inputs and the weights, adds a bias $b \in \mathbb{R}$ to this sum, and passes it through an activation function σ which can introduce some kind of a non-linearity to the computation. The whole mechanism is shown in Figure 2.1. The summation and the activation function application can be formally split as in Figure 2.2.

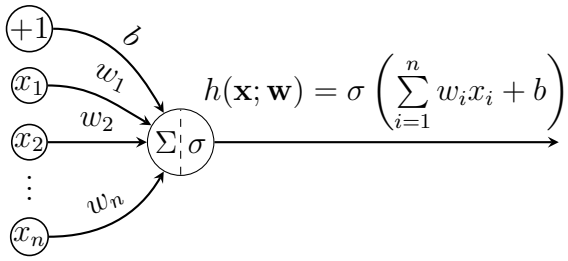


Figure 2.1: A model of a single neuron: $x_1, x_2, \ldots, x_n$ are inputs, $w_1, w_2, \ldots, w_n$ are weights, b is a bias, σ is an activation function.

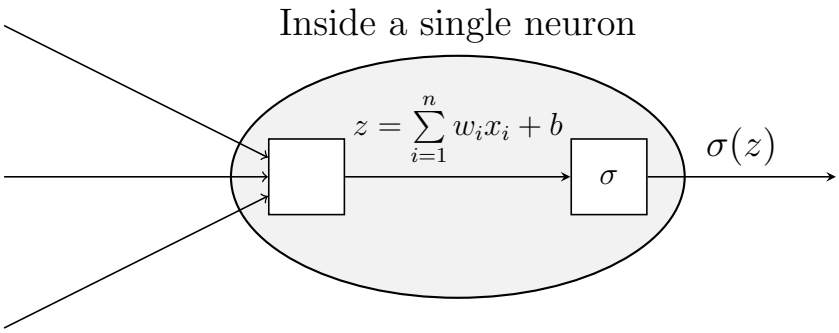


Figure 2.2: Formally splitting the neuron structure in two parts.

Defining $\mathbf{x}^T = [x_1, x_2, \ldots, x_n]$ and $\mathbf{w}^T = [w_1, w_2, \ldots, w_n]$ (by convention, $\mathbf{x}$ and $\mathbf{w}$ are column vectors), we can write $h(\mathbf{x}; \mathbf{w}) = \sigma(\mathbf{w}^T\mathbf{x} + b)$ for the neuron output. Popular choices for the activation function σ, together with their graphs in Figure 2.3, are:

- Identity: $\sigma(x) = x$
- Rectified linear unit (ReLU): $\sigma(x) = \max(0, x)$
- Logistic sigmoid: $\sigma(x) = \frac{1}{1+\exp(-x)}$
- Tanh: $\sigma(x) = \tanh(x)$

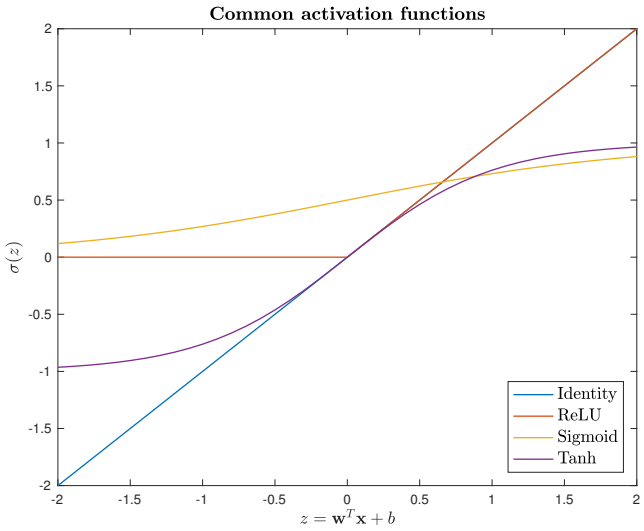


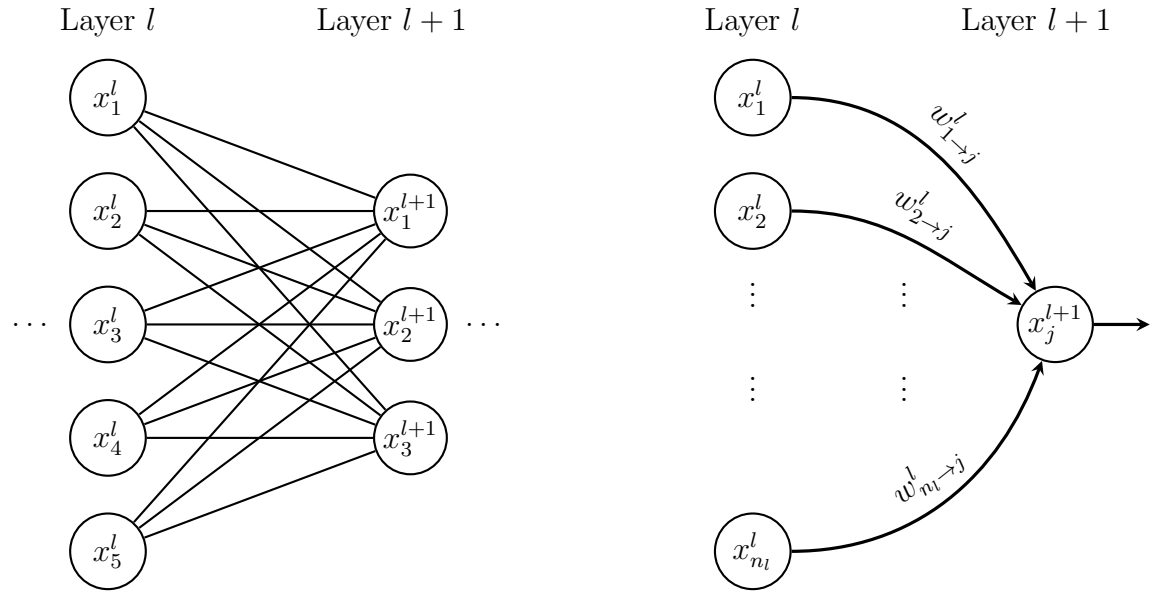
Figure 2.3: Activation functions graph.

The output of each neuron can become an input of any other neuron, resulting in arbitrary neural network architectures. I will only concentrate on feedforward ones, where the computation graph does not contain cycles. It is common to organise the neurons in layers, such that each layer processes inputs from previous layers. The next sections describe the different types of transitions between layers used in the project.

Neural network training will be discussed in greater detail later, but it is important to mention that our goal during training is to find such weights and biases which maximise the predictive power of the network on the training data.

Fully Connected Layers

As the name suggests, a fully connected neural network layer will consider all of its preceding layer's outputs as inputs for all of its neurons, as exemplified in Figure 2.4a.



(a) Layer l with 5 neurons fully connected to layer $l+1$ with 3 neurons.

(b) Notation with which I will refer to the layers, neuron values and weights connecting the neurons.

Figure 2.4: Fully connected layers.

Following the notation in Figure 2.4b, I will define a layer l to be the vector $(\mathbf{x}^l)^T = [x_1^l, x_2^l, \ldots, x_{n_l}^l]$, with n_l elements, and denote the weight of the edge between neuron i in layer l and neuron j in layer $l+1$ with $w_{i\to j}^l$. If we define the weight matrix connecting the two layers to be $\mathbf{W}_{ji}^l = w_{i\to j}^l$ with dimensions $n_{l+1} \times n_l$, from the definition of a single neuron it follows that

$$z_j^{l+1} = \sum_{i=1}^{n_l} w_{i\to j}^l x_i + b_j^{l+1} = \sum_{i=1}^{n_l} \mathbf{W}_{ji}^l x_i + b_j^{l+1} \tag{2.1}$$

$$x_j^{l+1} = \sigma(z_j^{l+1}) \tag{2.2}$$

where b_j^{l+1} is the added bias to neuron j in layer $l+1$ and z_j^{l+1} is its intermediate result. By extending the definition of the activation function so that it can be pointwise applied to vectors as well (i.e. $\sigma(\mathbf{z})_i = \sigma(z_i)$), the transition from layer l to layer $l+1$ can be easily described by the equation

$$\mathbf{x}^{l+1} = \sigma\left(\mathbf{W}^l\mathbf{x}^l + \mathbf{b}^{l+1}\right) \tag{2.3}$$

When training the parameters of a fully connected layer, we will have to fit $n_l n_{l+1}$ weights and n_{l+1} biases, resulting in $n_l n_{l+1} + n_{l+1} = (n_l + 1)n_{l+1} = O(n_l n_{l+1})$ parameters in total.

Convolutional Layers

As introduced in the Toy Problem from the Introduction chapter, I will eventually want my neural networks to be able to process information about images. In such cases the fully connected approach has two major drawbacks.

First, it does not scale well for large images. If each pixel of a 200×200 image is represented in a separate neuron, this results in a layer of size 40,000. Connecting multiple fully connected layers with comparable sizes consecutively becomes infeasible because of the quadratic number of parameters which will have to be fit during the training for each layer.

The second issue with fully connected layers is that the neurons in each layer and the weights connecting the layers are independent of each other. This ignores their relative spatial arrangement and redundancy that possibly exists in the image structure, missing the opportunity for weights sharing across space. If there is, say, a triangle in the upper left corner of an image, ideally we do not want the network to have to learn how to recognise it again next time when the triangle occurs in the lower right part of the picture. This observation about the redundancy in the learning process of fully connected layers gives us insight for constructing convolutional ones.

The idea of a convolution between an image and a kernel as a method for enhancing, emphasizing and extracting useful local features is classical for the Computer Vision field. When performing this operation a kernel (a small real-valued matrix, sometimes also called a filter) is slid over an image and their elements are pointwise multiplied. Then, the sum of the products is written to the output matrix (Figure 2.5).

Depending on the settings, sufficient zero padding can be added outside the border of the image if we want the operation to result in a matrix of the same size as the input. Further configurations are the horizontal and the vertical strides, determining the step size with which we slide the kernel. There exist complete tutorials on the clumsy details of convolutional arithmetic [11], which are omitted here for the clarity of the presentation. The effect of applying convolution with stride S and padding P is visualised only for a single dimension with size W in Figure 2.6[1].

In order to translate this mechanic to convolutional networks, we present each layer as a 3D volume of neurons with its respective width, height and depth. We can think of the width and the height as somewhat rescaled dimensions of the original image while

[1]The next two figures in this section were inspired from the clear presentation of Andrej Karpathy's course on Convolutional Neural Networks in Stanford University.

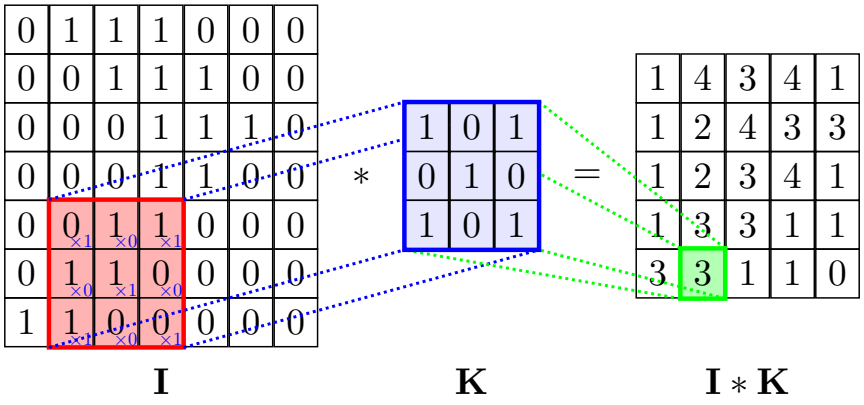
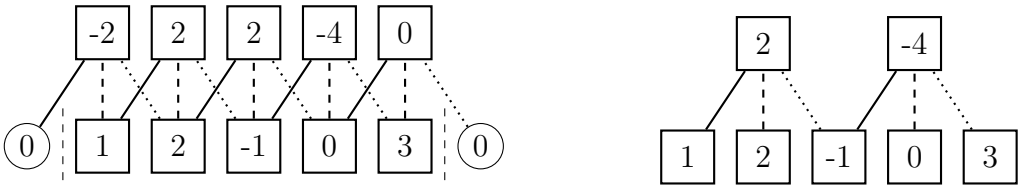


Figure 2.5: Result of the convolution of image **I** with kernel **K**.



(a) Stride $S = 1$, Padding $P = 1$.

(b) Stride: $S = 2$. Padding: $P = 0$.

Figure 2.6: Stride and padding effect to convolution when applied in a single dimension with size $W = 5$. The applied filter is $\boxed{1}\boxed{0}\boxed{-1}$ with size $F = 3$. Solid lines represent multiplication with +1, dashed – with 0, and dotted – with -1, empasising the idea of weight sharing across space.

the depth corresponds to the number of channels carrying useful information (e.g. in the input layer we have 3 channels for RGB images and 1 channel for grayscale images). It is possible to stack convolutional layers one after another (Figure 2.7), where the transitions to subsequent layers are determined by a set of weights, organised as 3D kernels and applied to the previous layer.

When applying a single kernel to a 3D layer with dimensions $w_1 \times h_1 \times d_1$, at any point of time while sliding it it covers only a small local area from the $w_1 \times h_1$ plane but spans through all the channels. That is, each filter must have dimensions $n \times n \times d_1$ (assuming they typically have a square shape in the width $\times$ height plane). Transition from a layer with dimensions $w_1 \times h_1 \times d_1$ to one with size $w_2 \times h_2 \times d_2$ requires d_2 such filters. The i-th filter produces results which go only to the i-th channel of the next layer (Figure 2.8).

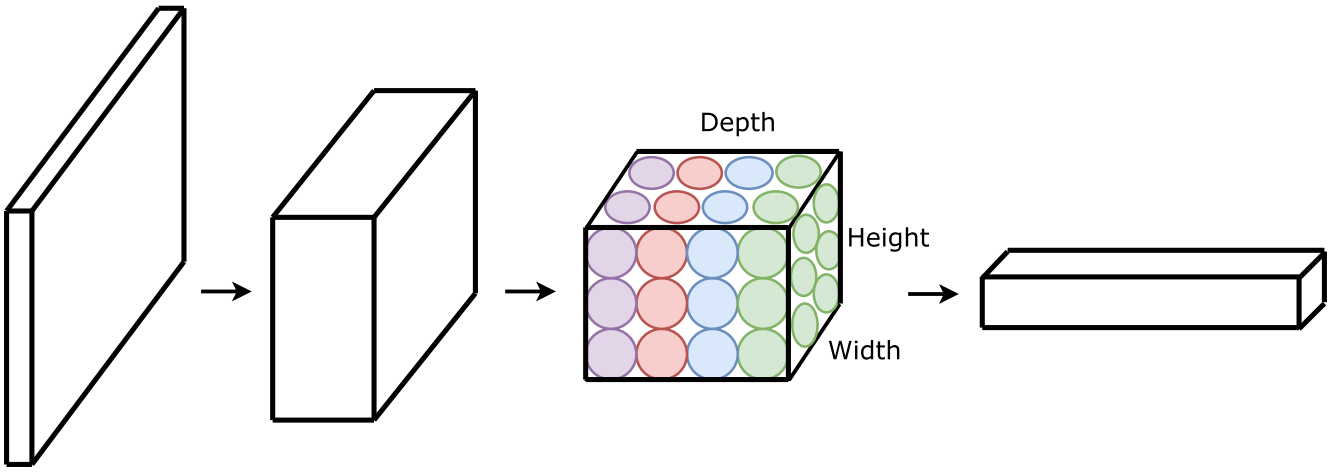


Figure 2.7: Stacked convolutional layers. Transitions to subsequent layers are determined by a number of filters slid across previous layers. Neurons from the same channels are in the same colour.

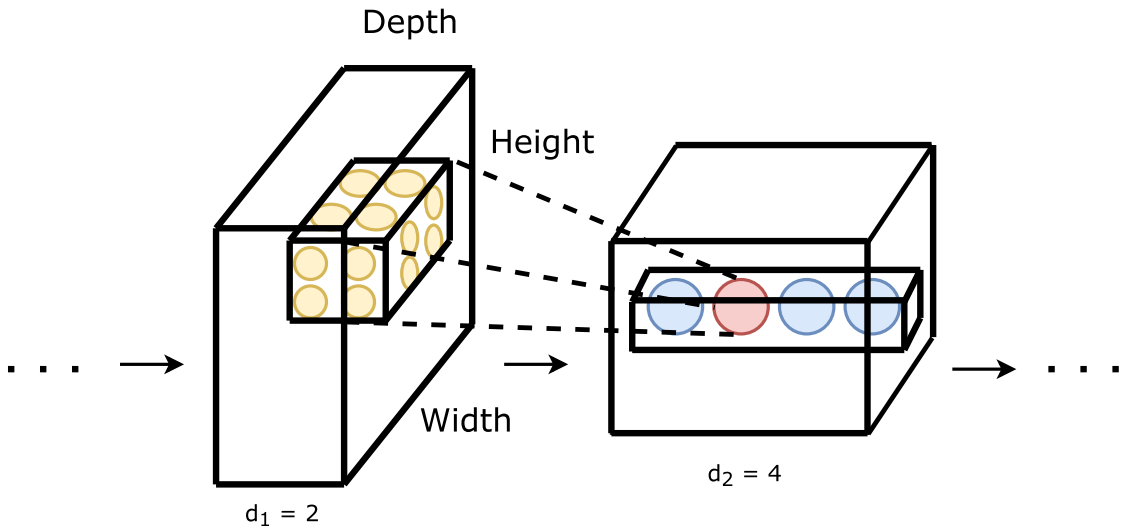


Figure 2.8: Applying a kernel to a 3D layer. The input layer on the left has depth $d_1 = 2$. Assume we want the kernel to cover area 2×2 from the width $\times$ height plane, hence the kernel dimensions must be $2 \times 2 \times 2$. The neurons covered by one such filter in a given position are in yellow. The depth of the output layer is set to $d_2 = 4$, so we will need 4 such filters. When applying the second of them to the yellow neurons, the result is a single neuron, denoted with red in the figure, sitting at its respective position in the second channel of the output layer on the right.

When stacking convolutional layers one after another, we are free to choose arbitrary values for their depths, but the width and height dimensions are restricted by the specifics of the 2D convolution in the width $\times$ height plane.

Just as with the simple 2D convolution in Figure 2.5, the dot product between the kernel and the volume it covers is taken. A bias is added and the sum is passed through

an activation function σ in order to obtain the final value of the neurons in the next layer. We can either use a separate bias for each neuron in the volume or share the same bias across the whole channel. Our goal when training a convolutional neural network is for the network to learn such kernels which are capable of extracting the most useful information out of the original image. When going from one layer to the next one, we have d_2 filters with size $n \times n \times d_1$, therefore the total number of parameters to fit is $d_2 \times n \times n \times d_1$ ($+d_2$ biases) and usually $n \ll w_1, h_1, w_2, h_2$. For comparison, if we wanted to connect the same number of neurons in a fully connected manner, this would have required $(w_1 h_1 d_1) \times (w_2 h_2 d_2)$ weights ($+w_2 h_2 d_2$ biases). The optimisation gain we achieve is of the order $O\left(\frac{w_1 h_1 w_2 h_2}{n^2}\right)$.

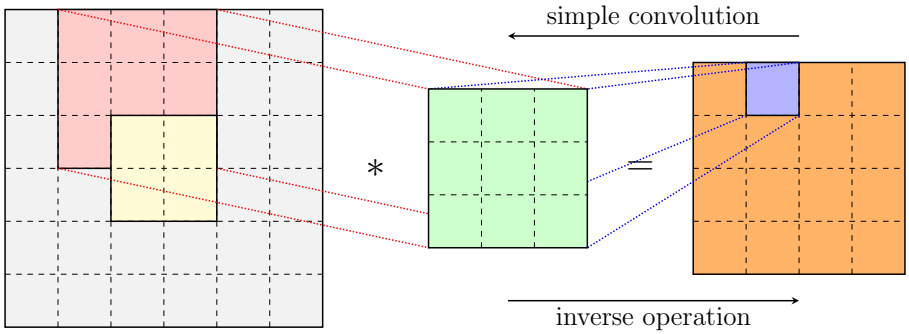
Intuitively, fully connected and convolutional architectures result in learning hierarchical representations. In the case of images for example, the first hidden layer may learn to detect edges, the next layer – more complicated shapes, and so on until we reach the final layer where we can either classify the original image or use the result as an input to another task. On the other hand, using transposed convolutional layers (also found in literature as deconvolutional ones), described in the next section, we can learn to reconstruct images starting from their high-level features.

Transposed Convolutional Layers

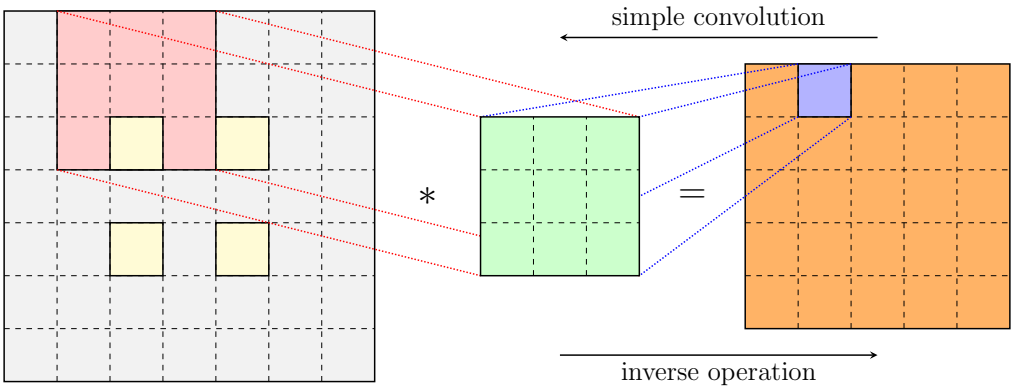
Transposed convolutional layers emerged out of the need to have an operation that is capable of inverting the effect of ordinary convolutional layers. Just as before, we present the layers as 3D volumes of neurons and the aim is again to learn filters which span the whole depth of the input layers when producing their outputs. The only difference from simple convolution, presented in the previous section, is in the way we slide the filters.

Setting the stride of an ordinary convolutional operation to $S > 1$ is capable of reducing the dimensions of the input image S times. With transposed convolutional layers we would like to achieve the opposite effect of upsampling – we would expect stride of S to increase the dimensions of the image S times.

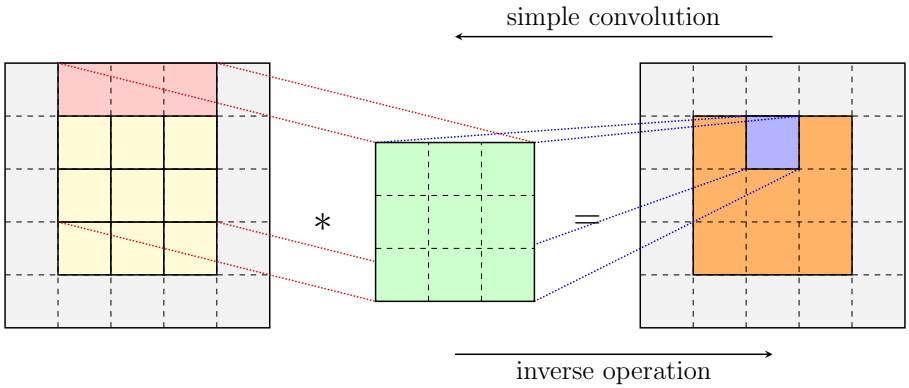
Examples of how transposed convolutional filters are being slid over the inputs are shown in Figure 2.9. The key observation is that after potentially making the input matrix sparser because of the upsampling we want to obtain, transposed convolution does not differ from the ordinary one and can be implemented with a simple convolutional operator. Apart from the stride S, the sparsity that should be added also depends on the desired shape of the output layer.



(a) Simple convolution of a 4×4 input (no padding) with a 3×3 kernel results in a 2×2 output. The inverse operation requires convolution of 6×6 input with (another) 3×3 kernel making unit strides.



(b) Simple convolution of a 5×5 input (no padding) with a 3×3 kernel using horizontal and vertical strides $S = 2$ results in a 2×2 output. The inverse operation requires convolution of 7×7 input (2×2 with a single zero inserted between the cells) with (another) 3×3 kernel making unit strides.



(c) Simple convolution of a 3×3 input (padded with $P = 1$ zeros around the borders), with a 3×3 kernel results in a 3×3 output. The inverse operation requires convolution of 5×5 input with (another) 3×3 kernel making unit strides.

Figure 2.9: Sliding a 3×3 kernel over transposed convolutional layers, intuitively acting as an inverse of ordinary convolution. The elements of the input layers are denoted with yellow, the kernel is in green and the output layers are in orange. Zero padding in the layers is denoted with grey. Fixing the kernel to a particular position in the input is coloured in red. The position of the resulting element when applying the kernel to the red area is denoted with blue.

2.1.1.2 Training Neural Networks. Backpropagation

To sum up, neural networks are computational graphs, in which the nodes (neurons) can be organised in layers and the nodes between two consecutive layers are connected with edges having a particular weight. How the layers are connected and what weight is assigned to each edge depends on which of the above connection schemes is used. For the purpose of this section, I will treat 3D convolutional layers as flattened one dimensional vectors. This nicely fits the theoretical framework I will present in this section but is an extra level of detail from an implementation perspective.

Building upon the definitions from the previous section, I will further introduce the following notation, visualised in Figure 2.10:

- Assume there are M layers in the network, $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^M$. Layer 1 is an input layer, layer M is an output layer and layers $2, \ldots, (M-1)$ are called hidden layers. It is perfectly possible to have no hidden layers at all.
- Set of T_l weights $\mathbf{\Omega}^l = \{\omega_1^l, \ldots, \omega_p^l, \ldots, \omega_{T_l}^l\}$ connecting layer l and layer $l+1$ $(1 \leq l < M)$. In convolutional and transposed convolutional layers these weights are shared across space. The set of all weights in the network is $\mathbf{\Omega} = \bigcup_{l=1}^{M-1} \mathbf{\Omega}^l$.
- Set of B_l biases $\mathbf{\Gamma}^l = \{\gamma_1^l, \ldots, \gamma_q^l, \ldots, \gamma_{B_l}^l\}$ added to neurons in layer l $(1 < l \leq M)$ just before passing the sum through an activation function. In convolutional and transposed convolutional layers it is possible to use the same bias for neurons from the same channel, so we can have some bias sharing as well. The set of all biases in the network is $\mathbf{\Gamma} = \bigcup_{l=2}^{M} \mathbf{\Gamma}^l$.
- $\boldsymbol{\theta} = \mathbf{\Omega} \cup \mathbf{\Gamma}$ – the set of all parameters defining the neural network computation.

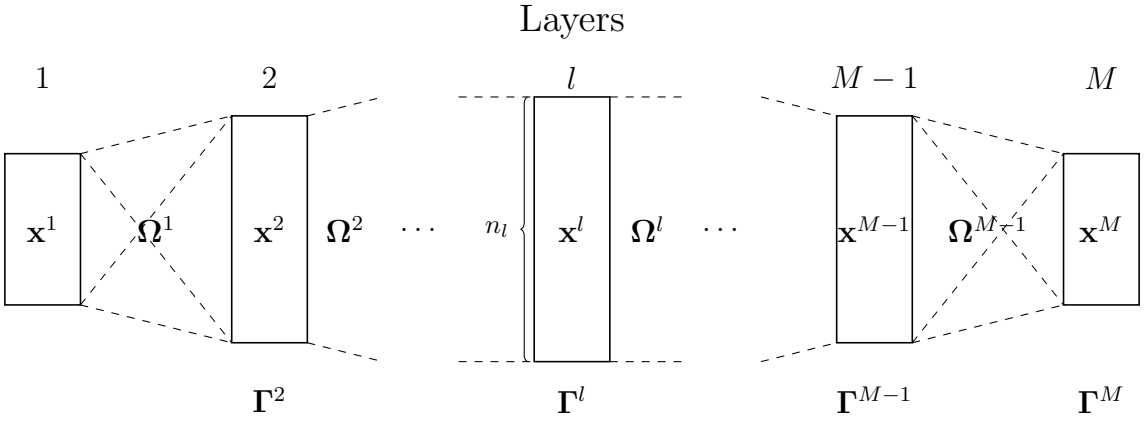


Figure 2.10: Schematic representation of a neural network with M layers. There are n_l neurons in layer l which is denoted with the vector $\mathbf{x}^l$. The set of weights defining the transition from layer l to layer $l+1$ is $\mathbf{\Omega}^l$. The set of biases added to any neuron from layer l before passing it through the activation function is $\mathbf{\Gamma}^l$.

With a particular neural network structure and some values for the parameters $\boldsymbol{\theta}$, we can define a function $\mathbf{y}' = f_{\boldsymbol{\theta}}(\mathbf{x})$ computed by the neural network. In order to obtain the

result of $f_{\boldsymbol{\theta}}(\mathbf{x})$, we simply put $\mathbf{x}$ at the input layer of the network, $\mathbf{x}^1$, perform the graph computation, defined by the network connections and the parameters $\boldsymbol{\theta}$, and return the output layer $\mathbf{x}^M$ as a result.

Neural networks are trained to maximise their predictive ability. For this purpose, we need a training set $\boldsymbol{X} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_N, \mathbf{y}_N)\}$ of N known (input, output) samples and a loss function $L(\mathbf{y}, \mathbf{y}')$ measuring how much the output of the neural network differs from the expected result for a single sample. An example of such a loss function which is classically used for regression problems is the squared error loss, $L(\mathbf{y}, \mathbf{y}') = \|\mathbf{y} - \mathbf{y}'\|^2$. As $\mathbf{y}'$ is a function of $\boldsymbol{\theta}$ and $\mathbf{x}$, we can write the loss function for a single sample as $\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) = L(\mathbf{y}, f_{\boldsymbol{\theta}}(\mathbf{x}))$. Combining the errors on all samples, we can define the loss function on the whole training set to be

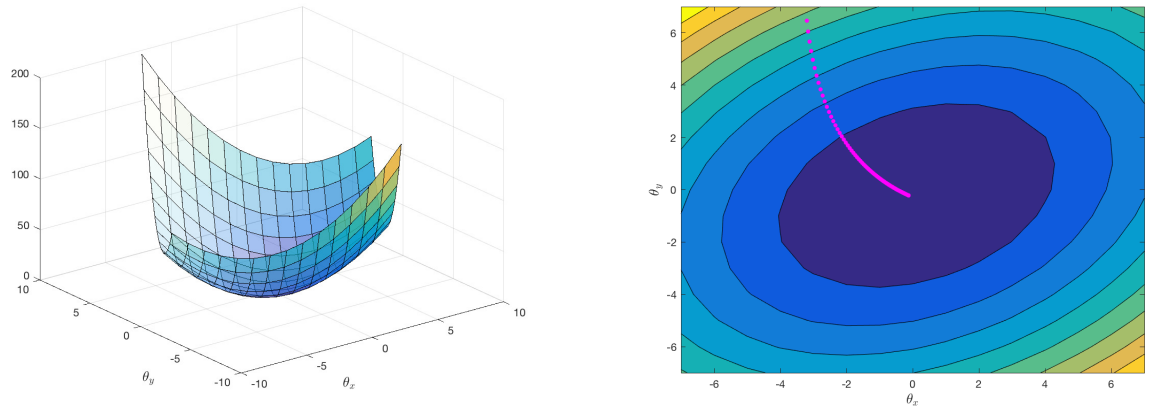
$$H(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\boldsymbol{\theta}; \mathbf{x}_i, \mathbf{y}_i) \tag{2.4}$$

When training a neural network, we want to find those values $\boldsymbol{\theta}$ for the parameters, which minimise $H(\boldsymbol{\theta})$.

The standard approach to solving this problem is applying gradient descent. Initially, the parameters $\boldsymbol{\theta}_0$ are chosen randomly. Then, small steps are taken, updating the parameters in the direction of the steepest decrease of $H(\boldsymbol{\theta})$ until convergence (Figure 2.11). That is,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda \frac{\partial H}{\partial \boldsymbol{\theta}}\bigg|_{\boldsymbol{\theta}_t} \tag{2.5}$$

where λ is known as learning rate. The difficult part is the computation of $\frac{\partial H}{\partial \boldsymbol{\theta}}$.



(a) 3D surface of the original function.

(b) 2D contour together with gradient descent steps (in red).

Figure 2.11: Gradient descent over a function of two variables.

Trying to simplify the partial derivative, we obtain

$$\frac{\partial H}{\partial \boldsymbol{\theta}} = \frac{\partial}{\partial \boldsymbol{\theta}} \left(\frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\boldsymbol{\theta}; \mathbf{x}_i, \mathbf{y}_i) \right) = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} \tag{2.6}$$

The backpropagation algorithm can now be used for the computation of $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$.

Utilising the weights notation introduced in the Fully Connected Layers section, it needs to be refined and extended in order to work for non-fully connected layers as well.

$$w_{i\to j}^{l} \equiv \begin{cases} \perp & \text{if } i \text{ and } j \text{ are not connected} \\ \omega_p^l & \text{for some } p \ (1 \le p \le T_l) \text{ otherwise} \end{cases} \tag{2.7}$$

Taking the idea of weight sharing in consideration, $w_{i\to j}^{l}$ is either undefined if i and j are not connected, or acts as a "pointer" to some weight from the weights set $\boldsymbol{\Omega}^l$ otherwise. Similarly,

$$b_i^l \equiv \gamma_q^l \quad \text{for some } q \ (1 \le q \le B_l) \tag{2.8}$$

The intermediate result z_j^{l+1} inside the structure of the neuron j in layer $l+1$ and the output x_i^l of neuron i in layer l can now be written as

$$z_j^{l+1} = \sum_{\substack{1 \le i \le n_l \\ w_{i\to j}^l \neq \perp}} w_{i\to j}^l x_i^l + b_j^{l+1} \tag{2.9}$$

$$x_i^l = \sigma(z_i^l) \tag{2.10}$$

Defining

$$\delta_i^l = \frac{\partial \mathcal{L}}{\partial z_i^l} \tag{2.11}$$

will be used as a proxy to computing $\frac{\partial \mathcal{L}}{\partial w_{i\to j}^l}$ (for $1 \le l < M$) and $\frac{\partial \mathcal{L}}{\partial b_i^l}$ (for $1 < l \le M$).

Running the backpropagation algorithm iteratively from layer M to layer 1 is established by the following 2 equations (fully derived in Appendix A.1).

$$\boxed{\delta_i^M = \frac{\partial \mathcal{L}}{\partial x_i^M} \cdot \sigma'(z_i^M)} \tag{2.12}$$

$$\boxed{\delta_i^l = \left(\sum_{\substack{1 \le j \le n_{l+1} \\ w_{i\to j}^l \neq \perp}} w_{i\to j}^l \delta_j^{l+1} \right) \sigma'(z_i^l)} \tag{2.13}$$

x_i^l and z_i^l can be computed doing a forward propagation in the graph, using the parameters $\boldsymbol{\theta}_t$. Equation (2.12) determines the partial derivatives with respect to the output layer neurons (the derivatives can be computed because the output layer variables directly participate in $\mathcal{L}$ and depend on its definition). The errors are then propagated back to the previous layers in the network according to Equation (2.13) – δ_i^l depends only on δ_j^{l+1} values, which are already known. The derivatives of the used activation functions in this project are put in Appendix A.2.

Using results (2.12) and (2.13), we can determine (full derivations again in Appendix A.1)

$$\boxed{\begin{aligned} \frac{\partial \mathcal{L}}{\partial w^l_{i\to j}} &= \delta^{l+1}_j x^l_i \qquad \text{for } 1 \le l < M \\ \frac{\partial \mathcal{L}}{\partial b^l_i} &= \delta^l_i \qquad \text{for } 1 < l \le M \end{aligned}} \tag{2.14, 2.15}$$

The only thing needed to obtain $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ is to relate these to $\frac{\partial \mathcal{L}}{\partial \omega^l_p}$ and $\frac{\partial \mathcal{L}}{\partial \gamma^l_q}$. From the summation rule we have

$$\frac{\partial \mathcal{L}}{\partial \omega^l_p} = \sum_{\substack{1\le i\le n_l \\ 1\le j\le n_{l+1} \\ w^l_{i\to j}\equiv\omega^l_p}} \frac{\partial \mathcal{L}}{\partial w^l_{i\to j}} \tag{2.16}$$

$$\frac{\partial \mathcal{L}}{\partial \gamma^l_q} = \sum_{\substack{1\le i\le n_l \\ b^l_i\equiv\gamma^l_q}} \frac{\partial \mathcal{L}}{\partial b^l_i} \tag{2.17}$$

which completes the derivation of the backpropagation algorithm in the general case where weight sharing is allowed.

2.1.2 Derivation of the Disentangled Autoencoder Framework

After covering the important details of neural networks and their training, what follows next is an introduction to autoencoders and the derivation of the disentangled autoencoder framework.

2.1.2.1 Introduction to Autoencoders

Autoencoders represent one of the earliest applications of neural networks to unsupervised learning (i.e. learning useful features of input data, $\mathbf{x} \in \mathbb{R}^n$, without corresponding labels). They consist of two building blocks (as illustrated by Figure 2.12):

- Encoder function – $encoder : \mathbb{R}^n \to \mathbb{R}^m$
  Takes the input $\mathbf{x}$ and maps it to an internal representation $\mathbf{z} = encoder(\mathbf{x})$, $\mathbf{z} \in \mathbb{R}^m$ (known as a code), written in one of the hidden layers.
- Decoder function – $decoder : \mathbb{R}^m \to \mathbb{R}^n$
  Takes the code $\mathbf{z}$ and creates a reconstruction $\mathbf{x}' = decoder(\mathbf{z})$, $\mathbf{x}' \in \mathbb{R}^n$ at the output layer of the autoencoder network.

Unlike the typical case where neural networks are trained to predict a certain target value given input $\mathbf{x}$, autoencoders' goal is to produce reconstructions as close as possible to the original inputs. This can be classically achieved by training all their parameters simultaneously to minimise the reconstruction error $L(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|^2$. Intuitively, when $n > m$ (i.e. $|\mathbf{x}| > |\mathbf{z}|$), the autoencoder is said to perform a dimensionality reduction and the vector $\mathbf{z}$ can be regarded as a compressed representation of $\mathbf{x}$.

When training an autoencoder, we want it to learn meaningful features about the input data and ideally to convey them in the code. Simply learning the identity function

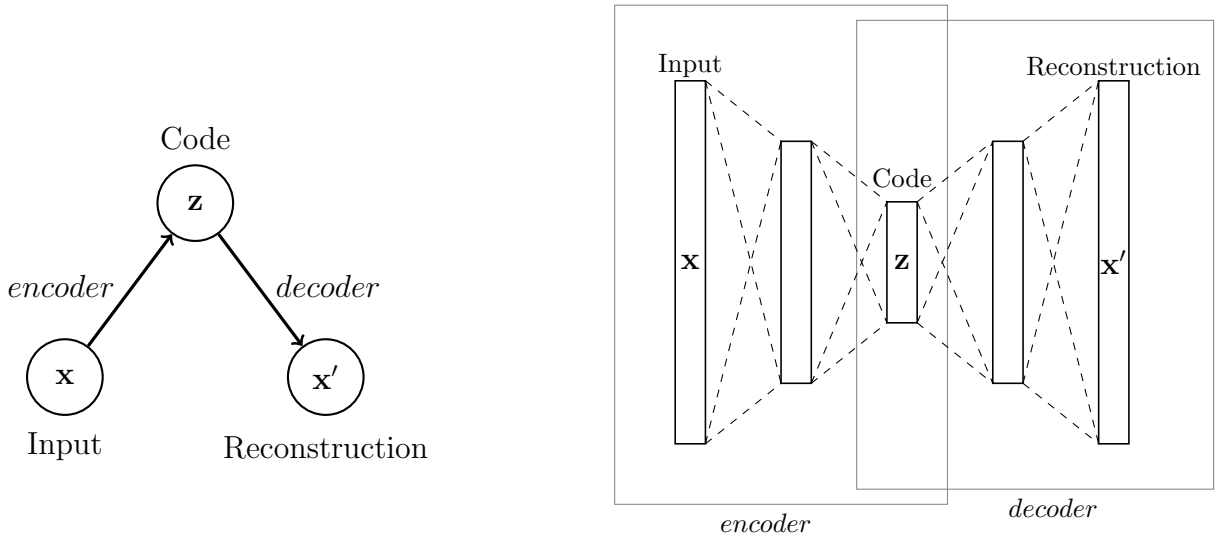


Figure 2.12: General structure of an autoencoder taking input $\mathbf{x}$ and building reconstruction $\mathbf{x}'$, going through the internal representation $\mathbf{z}$.

is unlikely to be useful if we want to do anything further than merely reducing the input size, such as building a generative model of our data. To that end, various regularisation techniques have been developed, encouraging the model to acquire further properties besides the ability to just reconstruct the input. One such approach is briefly discussed next.

Denoising Autoencoders

When training denoising autoencoders, random noise, governed by a given corruption process, is being injected to the input before passing it through the network. As the name implies, the goal of the denoising autoencoder is to reconstruct the original data and undo the corruption rather than to simply copy the input to the output. This idea is shown schematically in Figure 2.13.

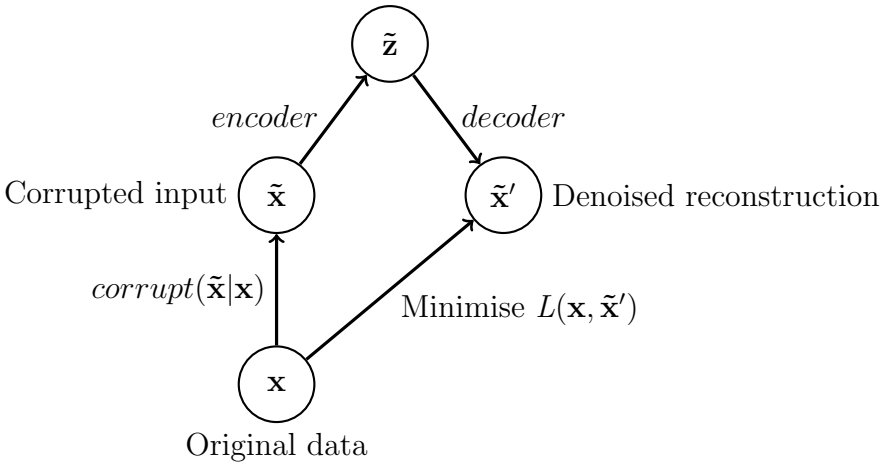


Figure 2.13: Denoising autoencoder overview. Various kinds of noise can be applied (e.g. for images: Gaussian, salt-and-pepper). The model is trained to minimise a predefined loss function L.

It has been shown that the autoencoder learns the structure of the distribution $p_{\text{data}}(\mathbf{x})$ as a useful by-product of denoising training [3, 2]. This insight inspires a probabilistic approach to autoencoders which will eventually enable us to derive the disentangled autoencoder loss function.

2.1.2.2 Disentangled Autoencoder Loss Function

To do so, we should first consider the problem of variational inference in general in greater detail.

Variational Inference

Variational Inference is useful for dealing with latent variable models. Assume we have a set of observations $\boldsymbol{X} = \{\mathbf{x}_i\}_{i=1}^N$ and the data was generated using some hidden, unobserved random variables $\mathbf{z}_i$, assigned for each observation using the probabilistic model p_θ. We can think of the process taking two steps:

1. A value $\mathbf{z}_i$ is drawn from the prior distribution $p_\theta(\mathbf{z})$.
2. $\mathbf{x}_i$ is then generated from the conditional distribution $p_\theta(\mathbf{x}|\mathbf{z})$.

In such cases $p_\theta(\mathbf{z})$ and $p_\theta(\mathbf{x}|\mathbf{z})$ are designed to be simple to compute, and from those we can compute $p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})$. However, when doing inference we are typically interested in calculating the posterior

$$p_\theta(\mathbf{z}|\mathbf{x}) = \frac{p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})}{p_\theta(\mathbf{x})} \tag{2.18}$$

which involves the expression for $p_\theta(\mathbf{x})$

$$p_\theta(\mathbf{x}) = \int_Z p_\theta(\mathbf{x}, \mathbf{z})\, \mathrm{d}\mathbf{z} = \int_Z p_\theta(\mathbf{x}|\mathbf{z})p_\theta(\mathbf{z})\, \mathrm{d}\mathbf{z} \tag{2.19}$$

Evaluating this integral requires spanning the whole space of $\mathbf{z}$ values and is often analytically intractable. As a result, it must be approximated. Stochastic approaches such as Markov Chain Monte Carlo algorithms might be too time consuming with large datasets, so we will concentrate on a deterministic method called Variational Bayes.

We introduce a recognition model $q_\phi(\mathbf{z}|\mathbf{x})$ as an approximation to the true intractable posterior $p_\theta(\mathbf{z}|\mathbf{x})$. How good this approximation is can be measured by the Kullback–Leibler divergence (Appendix A.3) of the two distributions $D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x}))$. The KL-divergence is always non-negative and becomes zero only when $q = p$ i.e. in the case where the approximation matches the true posterior exactly. Conversely, the greater the value is, the more the two distributions differ.

At the same time, when performing learning via maximum likelihood we want to maximise the log-likelihood of observing the data: $\log p_\theta(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \sum_{i=1}^{N} \log p_\theta(\mathbf{x}_i)$, assuming the observations are independent. By combining these two ideas, we can define the evidence lower bound (ELBO)

$$\text{ELBO}(\theta, \phi) = \log p_\theta(\mathbf{x}) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x})) \tag{2.20}$$

which, as the name suggests, is a lower bound of the model evidence (or log likelihood), $\log p_\theta(\mathbf{x}) \geq \text{ELBO}(\theta, \phi)$, with the equality being achieved when q approximates p perfectly. Thus, considering it as an optimisation problem, maximising ELBO with respect to θ and ϕ approximates maximum likelihood learning. It is shown in Appendix A.4 that this expression can be rewritten in the form

$$\text{ELBO}(\theta, \phi) = \underbrace{\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]}_{\text{Reconstruction accuracy}} - \underbrace{D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}))}_{\text{Regularisation}} \tag{2.21}$$

This is precisely the function maximised by the Variational Autoencoder of Kingma and Welling in [24].

Disentangled Autoencoder

Returning to the context of autoencoders, the encoder and the decoder can be made probabilistic – the encoder will refer to the recognition model $q_\phi(\mathbf{z}|\mathbf{x})$, trying to approximate $p_\theta(\mathbf{z}|\mathbf{x})$, while the decoder will implement the generative model $p_\theta(\mathbf{x}|\mathbf{z})$, as shown in Figure 2.14.

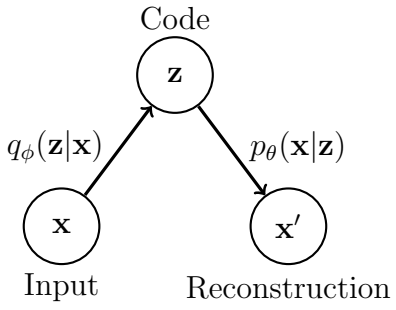


Figure 2.14: Probabilistic autoencoder overview. The code can be seen as sampled from the distribution $q_\phi(\mathbf{z}|\mathbf{x})$ and the reconstruction – generated from the code with probability $p_\theta(\mathbf{x}|\mathbf{z})$.

Analysing (2.21) more closely, we see that the first term of the RHS encourages the autoencoder to produce good reconstructions by maximising the probability of observing the data on average over all possible latents $\mathbf{z}$, while the second term acts as a regulariser, pushing the autoencoder towards learning distributions over latent variables $\mathbf{z}$ that are close to the prior $p_\theta(\mathbf{z})$. As already mentioned in the Introduction, the elements of the codes, produced by the autoencoder, are desired to eventually encode statistically independent features about the data separately, in different positions. This motivates the choice to set $p_\theta(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ since all components of the isotropic normal distribution are perfectly uncorrelated, which is in line with the aimed effect.

Extending these ideas further, a disentanglement parameter $\beta \geq 0$ is introduced. It controls the relative importance of the two kinds of errors, and in particular the level of learning pressure put on $q_\phi(\mathbf{z}|\mathbf{x})$ to approximate $p_\theta(\mathbf{z})$. Thus, the disentangled autoencoder maximises the objective function

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - \beta\, D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z})) \tag{2.22}$$

where $p_\theta(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$, by jointly optimising the encoder and decoder parameters, $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ respectively. This completes the description of the disentangled autoencoder framework.

An alternative, yet similar, approach to deriving the loss function (2.22), taken by Higgins *et al.* in [17], utilising constrained optimisation theory, can be found in Appendix A.5.

2.2 Requirements Analysis

After successful consolidation of the theoretical material, presented in the previous section, I could proceed with accomplishing the primary goal of the project in **implementing a disentangled autoencoder and evaluating the effects of the coefficient β on its performance**, as defined in the Project Proposal. In order to do so, the following deliverables were required (Table 2.1):

Requirement	Priority	Difficulty
Fully Connected Autoencoder	**High**	**High**
Convolutional Autoencoder	**Medium**	**High**
Denoising Support	**Medium**	**Low**
Disentangled Autoencoder Suite (combining the above three)	**High**	**Medium**
Dataset Management Module	**High**	**Medium**
Evaluation Suite	**High**	**Medium**
Experiment Runner	**High**	**Medium**
Visualisation Suite	**Medium**	**Low**

Table 2.1: Project requirements with their respective priorities and difficulties. High priority components are needed for the core project, the rest are desirable extensions.

Analysing their interaction with each other carefully (Figure 2.15), it became clear that reproducing the disentangled autoencoder structure was the most critical part of the project – it was a topic not explicitly covered by the Computer Science Tripos and at the same time was vital for the project evaluation. Therefore, in order to minimise the risk, it was decided to implement a fully connected architecture first, and then to add extensions if time permitting.

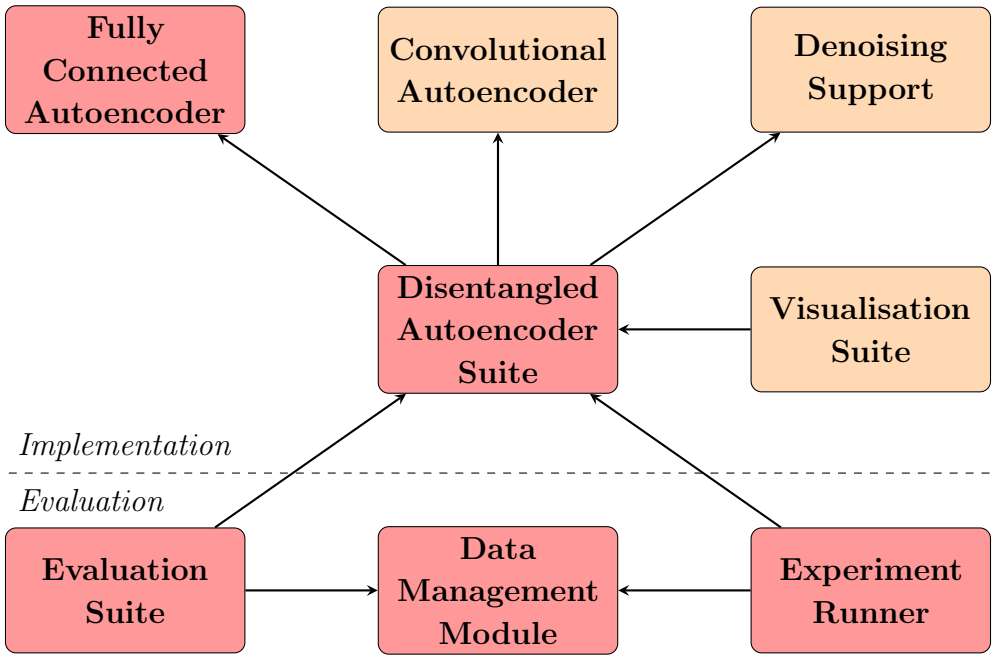


Figure 2.15: Interaction between the system components. $X \rightarrow Y$ indicates a dependency of X on Y. The modules required for implementation and evaluation are separated.

2.3 Choice of Tools

2.3.1 Programming Languages

Early on in my project I decided to use Python 2.7 as a primary development language. It is widely used by the deep learning research community, provides a huge number of libraries I could utilise and its scripting nature allowed me to partially automate the testing and evaluation process. For visualising and analysing the obtained experimental results I used MATLAB.

2.3.2 Libraries

Apart from the standard built-in Python libraries, the third-party software used in the project is listed in Table 2.2. The library of key importance is TensorFlow – its neural network functionalities have made it a first choice in deep learning research according to Andrej Karpathy's recent unofficial report[2]. It has its own computational model, with which I needed to get used to, and can be deployed on a GPU without any code changes.

[2] `https://medium.com/@karpathy/a-peek-at-trends-in-machine-learning-ab8a1085a106`

Library	Version	Purpose
TensorFlow	0.12	Neural networks mocking, training and testing
NumPy	1.12	Efficient mathematical operations with vectors and matrices; High precision double arithmetic
PIL	1.1.7	Python Imaging Library; Image manipulation
matplotlib	2.0.0	Image visualisation

Table 2.2: Libraries used during the project development.

2.3.3 Development Environment

The following hardware was used throughout the execution of the project:

- My own machine (macOS Sierra 10.12) for the majority of the project implementation, unit testing and dissertation writeup.
- MCS machine (Ubuntu 16.04 LTS) for the rest of the dissertation writing.
- NVIDIA TITAN X (Pascal) GPU (Ubuntu 16.04 LTS, 12 GB RAM), to which I was given access from the Computational Biology group at the Computer Laboratory, for the rest of the project implementation and the entirety of the evaluation.

Having to work on a remote server was an extra source of inconvenience which was dealt with by choosing a suitable range of tools which could easily run in the terminal. A complete list of the utilised tools can be found in Table 2.3.

Tool	Purpose
`vim 8.0`	Text editor
`git 2.10`	Revision control
`tmux 2.1`	Terminal multiplexer
`ssh`	GPU login
TeXstudio 2.12.4	LaTeX editor
PGF/Ti*k*Z 3.0	Graphics package
`draw.io`	Diagrams and graphics editor

Table 2.3: Tools used for the project development.

2.3.4 Backup Strategy

Special care was taken in order to protect myself against unexpected hardware and/or software failures. Developing the project on multiple machines was yet another reason to maintain revision control of my project and dissertation sources. My `git` repositories were hosted by GitLab and were regularly pulled from the workstations mentioned in the previous section. This guaranteed at any point of time my work was in at least two locations. Occasional uploads were done to my personal file space on Google Drive and external memory resources. The contingency plan is summarised in Figure 2.16.

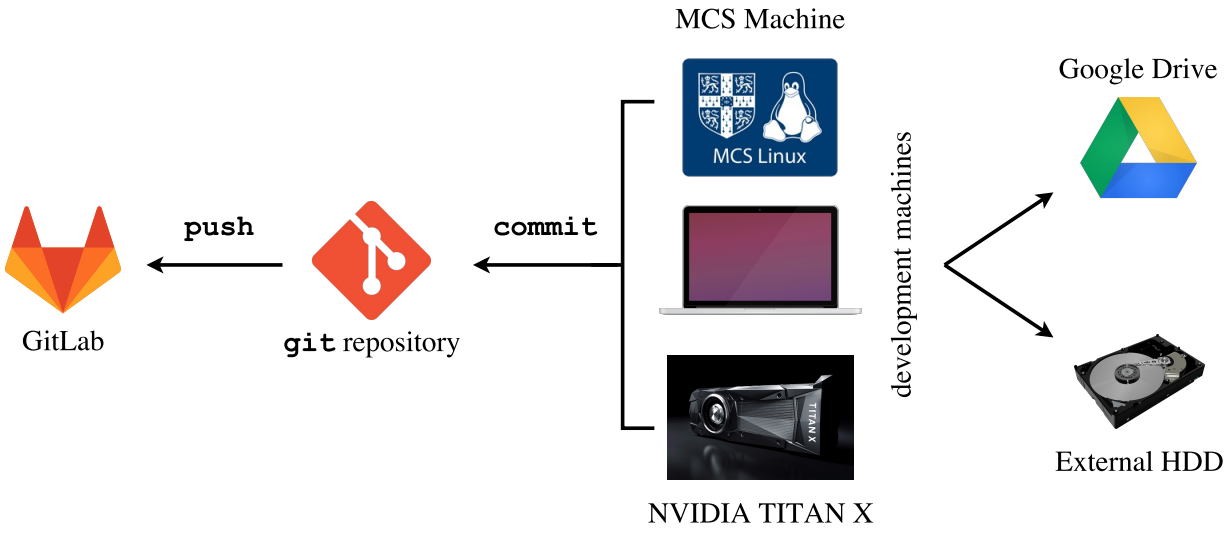


Figure 2.16: Overview of the backup strategy.

2.4 Software Engineering Techniques

A significant part of the theory presented in 2.1 was new to me and I did not have prior experience with any of the libraries listed in 2.3.2, so I decided on employing the Iterative Development Model as the software engineering model for the project. The system functionalities were incrementally developed and tested in iterations, analysing the state of the project after each step. In this way, the risks were minimised, leaving myself with enough time for reaction in case of unpredicted challenges from theoretical or implementation perspectives.

The code was modularised and logically organised – each source file contained a separate functional unit, allowing me to develop them independently. I adhered to good software practices, following Google's Python Style Guide[3] and writing comments and documentations for the most subtle parts of the code. Unit and integration testing infrastructure was built on my own development machine, operating on smaller datasets, in order to have more guarantees that the code I deploy and execute on the GPU is correct.

2.5 Summary

This chapter included a self-contained presentation of the theory behind neural networks and the details of the backpropagation algorithm used for training them. The special neural network structure of autoencoders was considered and the disentangled autoencoder framework was derived as a corollary of variational inference theory.

Software engineering aspects and practices were also taken into account in the project preparation. The use of the chosen languages and libraries was established and specifics of the development machines, the employed backup strategy and the project development lifecycle were given as well.

[3] `https://google.github.io/styleguide/pyguide.html`

In the next chapter I am going to present further details from the project implementation, the problems faced during this phase and will explain how they were resolved.

Chapter 3

Implementation

This chapter concenrs my own work towards reimplementing the necessary algorithms and experimental setup required for the successful execution of the project, utilising the third-party libraries, mentioned in the Preparation. The codebase was built from scratch and its final version contained more than 2,000 lines of code.

The implementation of the Data Management Module and the Disentangled Autoencoder Suite are described first. As will be seen later, the type of the processed data influences the design of the autoencoder output layer. TensorFlow, and its approach to building neural networks as computation graphs are briefly introduced. I pay particular attention to how the autoencoder is constructed so that the loss function derived in the Preparation chapter can be tractably approximated. The challenges encountered during the training process are emphasised and a thorough description of how they were resolved is given. The implementation details of the Evaluation Suite and the Experiment Runner will be continued in the Evaluation chapter.

3.1 Data Management Module

Before diving into the Disentangled Autoencoder Suite, I will first introduce the types of data which will eventually need to be processed. As asserted in the Introduction, I am closely following the work presented by Higgins *et al.* [17] in the development of my core project. However, to the best of my knowledge, the data used in their experiments is not open sourced, so I had to create my own synthetic dataset, ShapesSet, to show the models I built work as expected. Some of the project extensions were tested on the famous MNIST[1] dataset of handwritten digits [26], which will also be introduced in this section.

3.1.1 ShapesSet

Higgins *et al.* made the key assumption about the observed data that it should possess transform continuities in order to be able to find some regularity in it in an unsupervised manner. That is, *we assume the data is generated by factors of variation,* ***densely*** *sampled from their respective continuous distributions.*

[1] `http://yann.lecun.com/exdb/mnist/`

In accordance with the above, I have constructed ShapesSet, a synthetic dataset of 64x64 binarised images containing each a single shape (Figure 3.1). The following factors were chosen for the image generation:

- Shape: square (□), ellipse (⬭) or triangle (△).
- Position X (16 values) and Position Y (16 values).
- Scale (6 levels).
- Rotation (60 values over the $[0, \pi]$ range).

The PIL library was used for all image manipulations in the implementation of the GenerateImage function, assumed by Algorithm 1. The function maps generating factors to actual images.

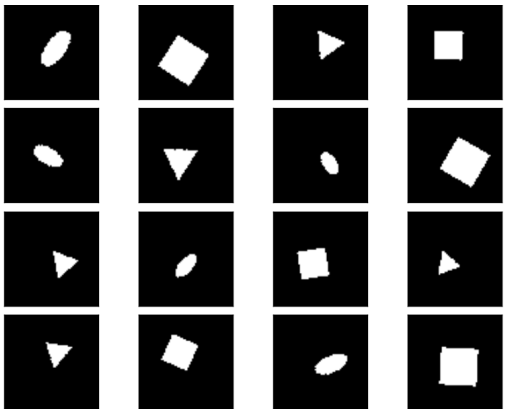


Figure 3.1: Examples of ShapesSet images revisited.

When running the experiments, the images were randomly separated in training, validation and test sets in a ratio 70:15:15 in a stratified way having approximately the same number of squares, ellipses and triangles in each subset. Special care was taken to reduce the leakage between the subsets by removing duplicate images incidentally caused by some idempotent transformations (Figure 3.2). The final dataset consists of 267,021 images. Algorithm 1 outlines the pseudocode for the data generation and splitting.

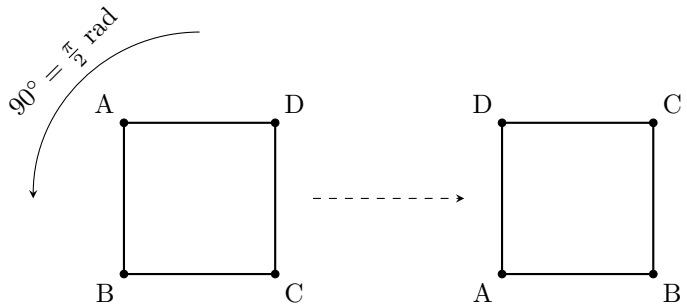


Figure 3.2: An example of a transformation that would produce the same image.

Algorithm 1 Generate ShapesSet

```
1: function SplitForShape(shape)
2:     images ← ∅
3:     for x ← 0 to POSITION_LIMIT do
4:         for y ← 0 to POSITION_LIMIT do
5:             for scale ← 0 to SCALE_LIMIT do
6:                 for rot ← 0 to ROTATION_LIMIT do
7:                     images ← images ∪ GenerateImage(shape, x, y, scale, rot)
8:                 end for
9:             end for
10:        end for
11:    end for
12:    return TrainTestValidationRandomSplit(images)
13: end function
14:
15: function GenerateShapesSet
16:    train, test, validation ← ∅, ∅, ∅
17:    for shape ← {□, ⬭, △} do                ▷ Ensure the sets are balanced:
18:        train′, test′, validation′ ← SplitForShape(shape)   ▷ Split separately
19:        train ← train ∪ train′                              ▷ for each shape
20:        test ← test ∪ test′
21:        validation ← validation ∪ validation′
22:    end for
23:    return train, test, validation
24: end function
```

3.1.2 MNIST

The MNIST dataset contains 70,000 grayscale images of hand-written digits (28×28 pixel), examples of which are included in Figure 3.3. They have been used as research benchmarks for the past 20 years so no work on their normalisation or splitting in train, test and validation subsets was required.

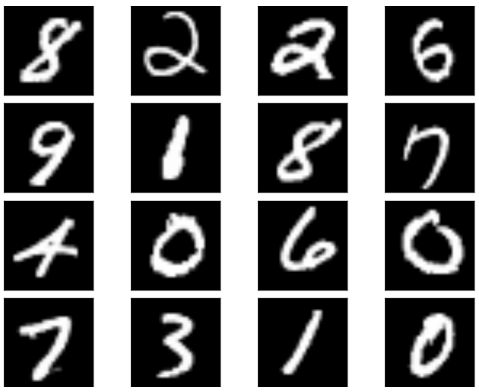


Figure 3.3: Examples of MNIST images.

3.2 Disentangled Autoencoder Suite

3.2.1 Overview

As emphasised in Section 2.2, the Disentangled Autoencoder Suite is the central component of this project. In order to have a greater flexibility in running various types of experiments during the Evaluation phase, I decided to encapsulate most of its functionalities, providing an API similar to that of established machine learning libraries running in Python, such as `scikit-learn` [28, 6] or `tflearn`. The suite was developed in multiple iterations and I eventually decided to split its base methods from those responsible for building the models' structure according to the desired autoencoder architecture (Figure 3.4). TensorFlow was heavily utilised for dealing with most of the simple neural network operations. Had I implemented them from scratch, it would have taken me significantly longer in order to achieve numerical precision and scalability comparable to those provided by the library. In fact, this could have been a whole project on its own. A brief high level summary of TensorFlow's specific computation model, mentioned in the previous chapter, is given next.

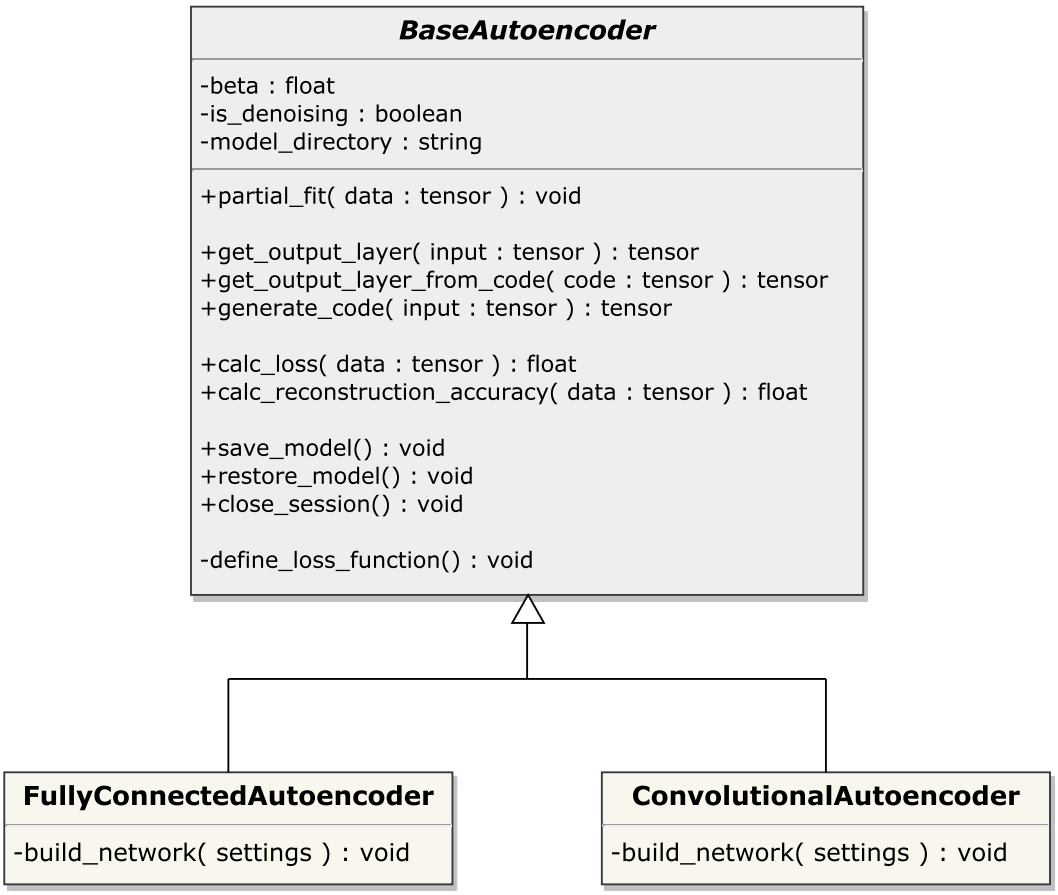
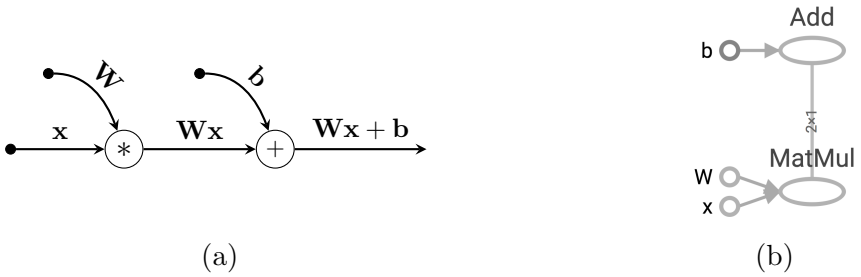


Figure 3.4: UML diagram of the autoencoder suite.

TensorFlow Computation Model

TensorFlow performs numerical computations on explicitly predefined directed graphs. The nodes contain mathematical operations and there are tensors (typed multi-dimensional arrays) "flowing" through the edges, acting as inputs to the operation nodes (Figure 3.5). Compared to a typical programming language model, we can think of TensorFlow as omitting the lexing stage and directly requiring the abstract syntax tree of the "program", defined by the graph, to be provided manually.



(a) (b)

Figure 3.5: $\mathbf{Wx} + \mathbf{b}$ represented in a graph form. (b) is the same expression, as visualised by TensorBoard, a tool used for visualising TensorFlow graphs.

In spite of the implementation overhead, this approach has two main advantages. First of all, in the context of neural networks, taking a particular computation graph, calculating the function L in its final node, a "backwards" graph can be defined, computing the derivative of L with respect to any parameter taking part in the computation, by using the chain rule (Figure 3.6). It was established in Section 2.1.1.2 that this is all we need in order to run backpropagation. The second advantage of the graph structure is that it allows for a high level of parallelism – both in performing individual operations in the nodes efficiently, utilising the vectorised format of the data, and in executing operations from multiple nodes simultaneously when possible, as the dependencies between them are made clear in advance.

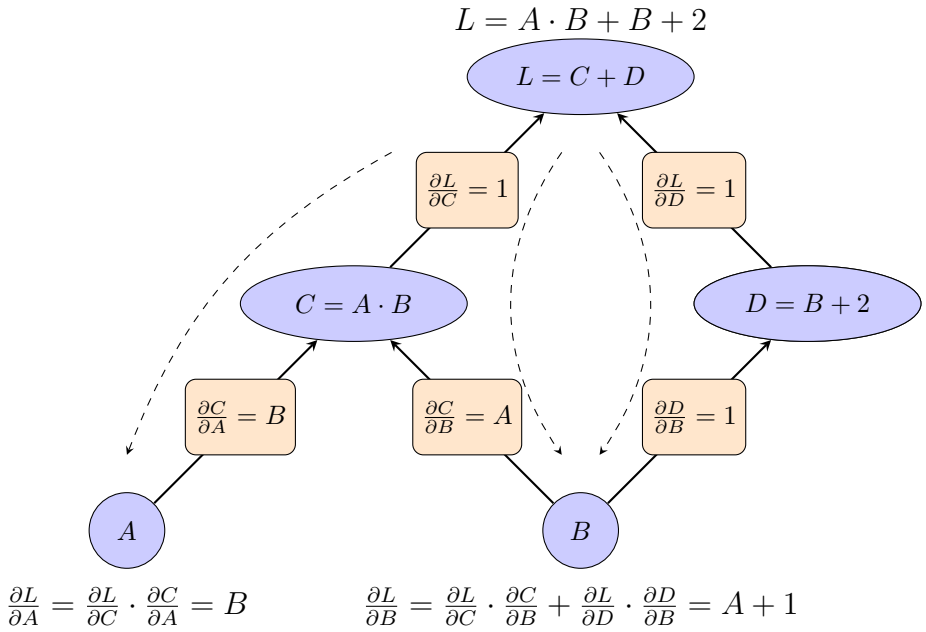


Figure 3.6: Calculating the partial derivatives of L with respect to A and B, using the chain rule. If we have an edge $X \to Y$, where X is an input to a node, whose operation results in Y, we can easily evaluate $\frac{\partial Y}{\partial X}$. The node operation is known, so there is a formula for its derivative. Moreover, the values of all variables, appearing in the final expression can be derived by going through the "forward" graph (in solid arrows). By following the dashed arrows (the "backwards" graph) backpropagation is executed.

TensorFlow programs, applied to neural network problems, typically follow the structure shown in Figure 3.7. In the next sections I will discuss each of these three stages from the disentangled autoencoder implementation in detail.

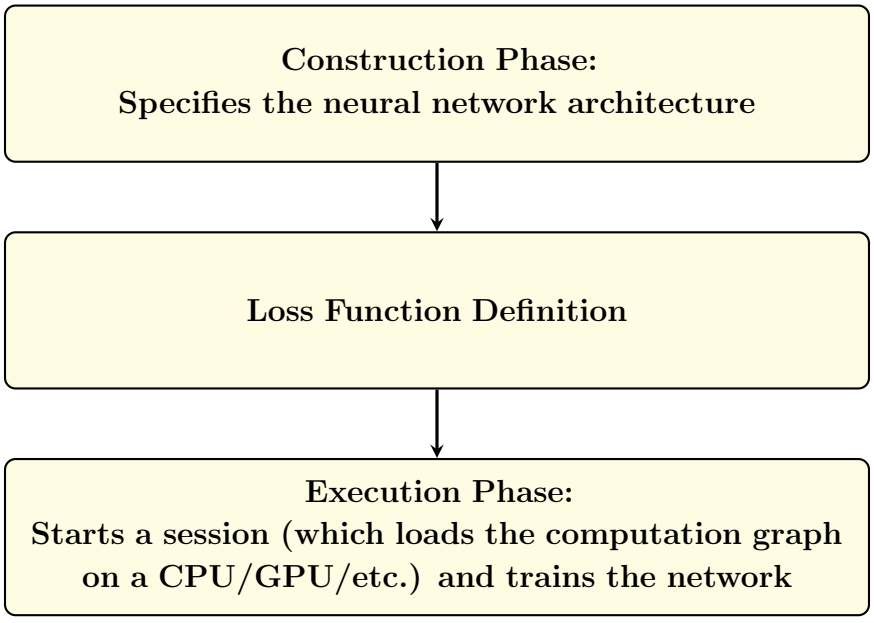


Figure 3.7: General structure of TensorFlow programs implementing neural networks.

3.2.2 Construction Phase

It was stated in Section 2.1.1 that the neural networks are organised by stacking layers one after another. The input layer of a fully connected autoencoder treats image pixels independently and represents them as a flattened 1D vector (of size $64 \times 64 = 4096$ for ShapesSet and $28 \times 28 = 784$ for MNIST). The input layer of a convolutional autoencoder contains original images represented as a 3D volume of neurons with just one channel as I only have binary or grayscale images (so the dimensions of the input 3D volumes are 64×64×1 for ShapesSet and 28×28×1 for MNIST).

Listing 1 provides a simplified source code exemplifying the general approach to constructing fully connected layers in the architecture, implementing the pipeline visualised in Figure 3.8. To stack convolutional or transposed convolutional layers, the matrix multiplication block is substituted with the respective operation (implemented in TensorFlow by `tf.nn.conv2d` or `tf.nn.conv2d_transpose`). The logistic and tanh non-linearities are provided by `tf.nn.sigmoid` and `tf.nn.tanh`.

It can be observed that TensorFlow takes off the complexity of managing the weights and biases and lets us work in one level of abstraction higher, concentrating on the models' specifications. Everything that follows in the subsequent sections builds on the pipeline presented here.

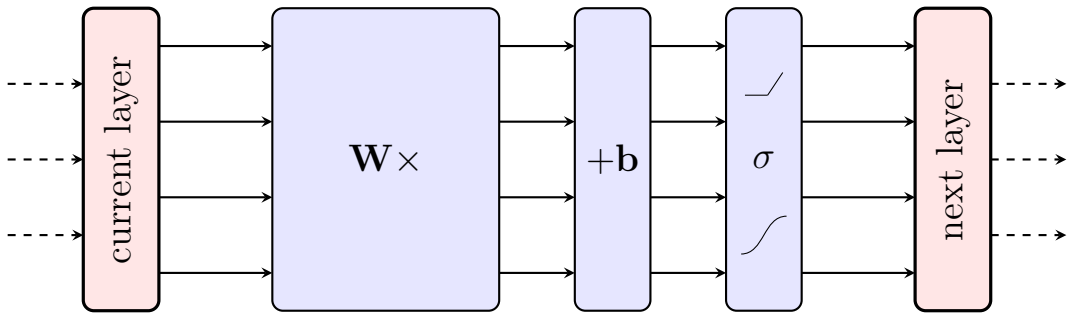


Figure 3.8: Neural network operations pipeline, as already introduced in the Preparation.

Listing 1 A simplified implementation of the pipeline in Figure 3.8 with TensorFlow.

```
import tensorflow as tf

def stack_fully_connected_layer(autoencoder, next_layer_size):
  fan_in = autoencoder.current_layer.size()
  fan_out = next_layer_size
  W = get_weights(shape=(fan_out, fan_in))
  b = get_bias(shape=(fan_out))
  autoencoder.current_layer = tf.nn.relu(
    tf.nn.bias_add(tf.matmul(W, autoencoder.current_layer), b))
```

3.2.3 Loss Function Definition

In the Preparation chapter, it was established that the disentangled autoencoder should maximise

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - \beta\, D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z})) \tag{2.22}$$

with $p_\theta(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ and β fixed. In this section I will present how the neural network can be constructed to evaluate this loss function so that the weights and biases can afterwards be trained via backpropagation. As defined previously, I will use $|\mathbf{x}| = n$ and $|\mathbf{z}| = m$ for the dimensions of the input/output layers of the autoencoder and the code respectively.

Calculating the Regularisation $-D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z}))$. Code Construction

While $p_\theta(\mathbf{z})$ is fixed, there are no constraints imposed on the form of $q_\phi(\mathbf{z}|\mathbf{x})$ for the moment. Taking into account the fact its KL-divergence with respect to the isotropic unit Gaussian normal should be minimised (in order to maximise $\mathcal{L}$), it is reasonable to make the simplifying assumption that $q_\phi(\mathbf{z}|\mathbf{x})$ is also approximately Gaussian normal with a diagonal covariance matrix, $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2\mathbf{I})$. Now $-D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z}))$ can be calculated analytically [24]:

$$-D_{KL}\left(\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2\mathbf{I})||\mathcal{N}(\mathbf{0}, \mathbf{I})\right) = \frac{1}{2}\sum_{j=1}^{m}\left(1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2\right) \tag{3.1}$$

To implement this, I made the neural network learn the means and the standard deviations of the elements of the code. Two intermediate fully connected layers were added, one for $\boldsymbol{\mu}$ and one for $\boldsymbol{\sigma}^2$, just before the code layer of the autoencoder and each element z_j of the code was sampled from $\mathcal{N}(\mu_j, \sigma_j^2)$, as outlined in Figure 3.9.

Sampling to Calculate $\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$

Rewriting $\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$ as

$$\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] = \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \log p_\theta(\mathbf{x}|\mathbf{z})\, \mathrm{d}\mathbf{z} \tag{3.2}$$

it becomes clear that the expression is intractable as $q_\phi(\mathbf{z}|\mathbf{x})$ and $p_\theta(\mathbf{x}|\mathbf{z})$ are generated from the autoencoder neural network. It is hard to solve the integral analytically but it can be approximated via sampling from the distribution $q_\phi(\mathbf{z}|\mathbf{x})$:

$$\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] \approx \frac{1}{S}\sum_{i=1}^{S} \log p_\theta(\mathbf{x}|\mathbf{z}_i) \tag{3.3}$$

where each $\mathbf{z}_i$ is generated as described in the previous section and visualised in Figure 3.9, and S is the number of samples drawn.

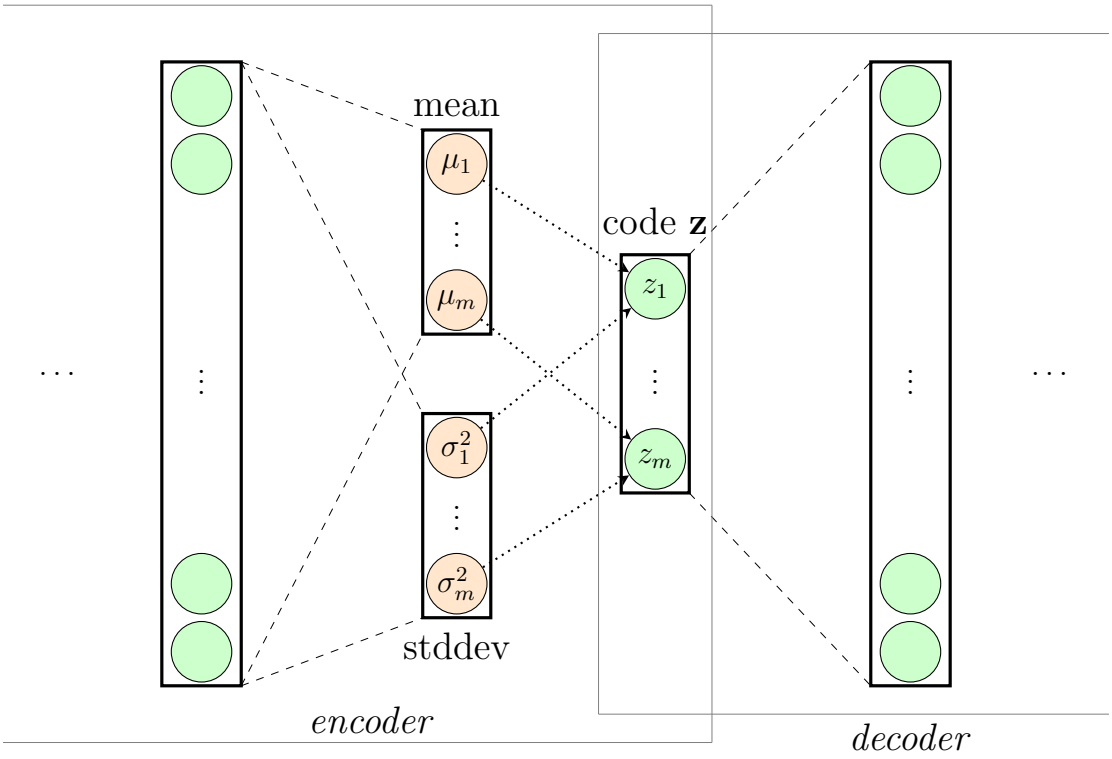


Figure 3.9: Learning the means and the standard deviations of the elements of the code. Dotted arrows indicate sampling, i.e. $z_j = \mathcal{N}\left(z_j|\mu_j, \sigma_j^2\right)$.

Minibatches

Taking this idea further, Kingma and Welling [24] found that if the original dataset $\boldsymbol{X} = \{\mathbf{x}_i\}_{i=1}^N$ of N samples is randomly split in batches of large enough size R (e.g. $R = 100$), the number of samples required per datapoint can be set to $S = 1$. Putting it all together, the average loss per example from a single batch, $\boldsymbol{X}^R$, is

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{X}_{avg}^R) = \frac{1}{R} \sum_{\mathbf{x} \in \boldsymbol{X}^R} \left(\log p_\theta(\mathbf{x}|\mathbf{z}) + \frac{\beta}{2} \sum_{j=1}^{m} \left(1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2\right) \right) \tag{3.4}$$

The next step is to see how $\log p_\theta(\mathbf{x}|\mathbf{z})$ can be formulated in the implementation framework that has been built so far.

Evaluating $\log p_\theta(\mathbf{x}|\mathbf{z})$. Output Layer Construction

Given that the data is predominantly binary, the autoencoder can be designed to produce Bernoulli outputs by choosing the activation function for the output layer of the decoder to be the logistic sigmoid. Its range is $[0, 1]$, so the values of the output layer can easily be interpreted as both grayscale levels and as probabilities of observing black or white in the respective pixels.

Let $decoder_\theta(\mathbf{z}) = \mathbf{x}'$ be the output layer of the autoencoder, so $p_\theta(\mathbf{x}|\mathbf{z}) = p(\mathbf{x} = \mathbf{x}'|\boldsymbol{\theta}, \mathbf{z})$.

This configuration resembles a binary classification problem in which

$$p_\theta(x_j = C|\mathbf{z}) = \begin{cases} x'_j & \text{if C} = 1 \\ 1 - x'_j & \text{if C} = 0 \end{cases} \tag{3.5}$$

for each pixel of the output. In the discrete case for x_j, this can be expressed in the form

$$p_\theta(x_j|\mathbf{z}) = (x'_j)^{x_j}(1 - x'_j)^{(1-x_j)} \tag{3.6}$$

Making the simplifying assumption that individual pixels can be treated independently of each other, it is obtained that

$$\log p_\theta(\mathbf{x}|\mathbf{z}) = \sum_{j=1}^{n} \log p_\theta(x_j|\mathbf{z}) = \sum_{j=1}^{n} x_j \log x'_j + (1 - x_j) \log(1 - x'_j) \tag{3.7}$$

which is simply a special case of cross-entropy. Plugging it back to Equation (3.4), a differentiable final closed form of the loss function is obtained

$$\boxed{\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{X}^R_{avg}) = \frac{1}{R} \sum_{\mathbf{x} \in \boldsymbol{X}^R} \left(\sum_{j=1}^{n} \left(x_j \log x'_j + (1 - x_j) \log(1 - x'_j) \right) + \frac{\beta}{2} \sum_{j=1}^{m} \left(1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2 \right) \right)} \tag{3.8}$$

where for each $\mathbf{x} \in \boldsymbol{X}^R$, provided as an autoencoder input, the encoder generates $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$, the code is sampled, $\mathbf{z} = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$, and the decoder takes the code and produces $\mathbf{x}'$ (using the sigmoid activation function for the output layer).

In the next section, partial derivatives of $\mathcal{L}$ are derived and the details behind the autoencoder training are discussed.

3.2.4 Autoencoder Training

Backpropagation Revisited

The formulae used for propagating the error backwards from the output layer through the hidden layers to the input layer were already derived in the Preparation chapter. However, the loss function node should be "linked" to the neurons directly participating in its expression in order to get the backpropagation algorithm started.

Denote with $\mathcal{L}'$ the loss for an individual data point (the expression inside the sum over $\boldsymbol{X}^R$ in Equation (3.8)). For the k-th neuron x'_k of the output layer we have:

$$\frac{\partial \mathcal{L}'}{\partial x'_k} = \frac{x_k}{x'_k} - \frac{1 - x_k}{1 - x'_k} \tag{3.9}$$

The partial derivatives of the mean and the standard deviation of the k-th element of the code are:

$$\frac{\partial \mathcal{L}'}{\partial \mu_k} = -\beta \mu_k \tag{3.10}$$

$$\frac{\partial \mathcal{L}'}{\partial \sigma_k^2} = \frac{\beta}{2} \left(\frac{1}{\sigma_k^2} - 1 \right) \tag{3.11}$$

Reparametrisation Trick

The only issue left is to make sure backpropagation can go through the random sampling of the elements of the code $z_j \sim \mathcal{N}(\mu_j, \sigma_j^2)$. For this purpose, an auxiliary random node, $\varepsilon_j \sim \mathcal{N}(0,1)$, is introduced for each code element and z_j is represented as $z_j = \mu_j + \varepsilon_j \sigma_j$, preserving its desired probability distribution (Figure 3.10). Then $\frac{\partial z_j}{\partial \mu_j} = 1$ and $\frac{\partial z_j}{\partial \sigma_j} = \varepsilon_j$, which allows for the error to be backpropagated through the code layer.

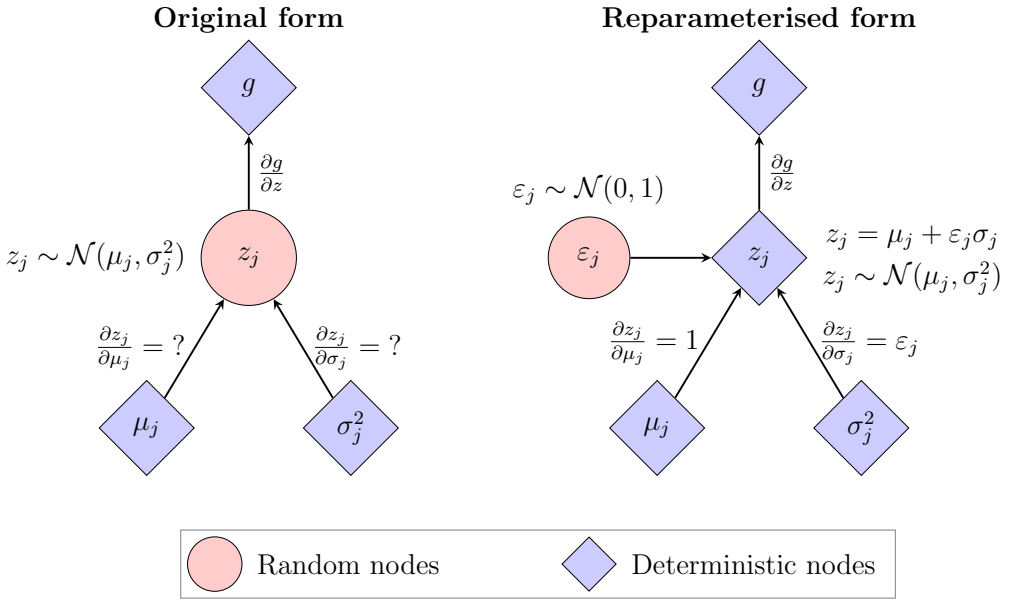


Figure 3.10: Random sampling operators.

This concludes the neural network design details required for approximating the loss function defined in the Preparation section. After ensuring all necessary partial derivatives can be computed, they are used to fit the model parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ (consisting of the neural network weights and biases) via a form of stochastic gradient descent, which is described next.

Adam Optimiser. Early Stopping

Thus far, the loss function $\mathcal{L}$ was constructed in a tractable way so that I can proceed with the objective to maximise it. In order to be consistent with literature where a function is typically being minimised, I will define $\mathcal{H} = -\mathcal{L}$.

$\mathcal{H}$ is minimised with respect to the whole dataset $\boldsymbol{X}$, not just for a single example $\mathbf{x}_i$. Precisely, we are looking for

$$\arg\min_{\theta,\phi} \mathcal{H}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{X}_{avg}) = \arg\min_{\theta,\phi} \frac{1}{N} \sum_{i=1}^{N} \mathcal{H}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}_i) \tag{3.12}$$

The idea of processing the data examples in batches instead of individually was already justified above. Apart from providing a way to avoid unnecessary sampling, this strategy also reduces the variance of the gradient updates, leading to faster convergence. Another

crucial point is that the individual gradients obtained for a single data point in the batch can be computed in parallel by TensorFlow (providing greater throughput and immensely utilising the speed-up guaranteed by GPUs) and then their average is taken in the end. The reason why the whole dataset is not treated as a single batch is predominantly the memory limit imposed by the RAM of the GPU – all of the data cannot fit in at once so the dataset is split in minibatches as we will see shortly.

Adam [23] was the optimisation algorithm chosen to govern parameters updates in each gradient descent iteration. It keeps track of the running averages of both the gradients and the second moments of the gradients (Algorithm 2). TensorFlow's built-in implementation of the algorithm once again facilitated the project execution.

Algorithm 2 The set of equations, describing Adam's optimisation step. $\mathbf{g}_t^2 = \mathbf{g}_t \odot \mathbf{g}_t$ indicates element wise square. All operations on vectors are element wise as well. β_1^t and β_2^t are β_1 and β_2 to the power t. The default values $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ were used. The learning rate was set to $\alpha = 0.001$, except for the case when training the autoencoder on ShapesSet with disentanglement parameter $\beta = 0$. Then it had to be further reduced to $\alpha =$ 3e-5 to ensure the numerical stability of the training process.

Require: α – learning rate
Require: $\boldsymbol{\theta}_0, \boldsymbol{\phi}_0$ – randomly initialised model parameters
$\mathbf{m}_0 \leftarrow \mathbf{0}$, $\mathbf{v}_0 \leftarrow \mathbf{0}$ ▷ Initial first and second moment vectors
$t \leftarrow 0$ ▷ Initialise Adam Optimiser timestep
procedure PARTIALFIT(*batch*) ▷ Adam iteration step for a given input minibatch
    $t \leftarrow t + 1$
    $\mathbf{g}_t \leftarrow \nabla_{\theta,\phi}\mathcal{H}(\boldsymbol{\theta}_{t-1}, \boldsymbol{\phi}_{t-1}; batch)$ ▷ Gradients at timestep t for *batch* via backprop
    $\mathbf{m}_t \leftarrow \beta_1 \cdot \mathbf{m}_{t-1} + (1 - \beta_1) \cdot \mathbf{g}_t$ ▷ Update biased first moment estimate
    $\mathbf{v}_t \leftarrow \beta_2 \cdot \mathbf{v}_{t-1} + (1 - \beta_2) \cdot \mathbf{g}_t^2$ ▷ Update biased second moment estimate
    $\widehat{\mathbf{m}}_t \leftarrow \mathbf{m}_t / (1 - \beta_1^t)$ ▷ Bias corrected first moment estimate
    $\widehat{\mathbf{v}}_t \leftarrow \mathbf{v}_t / (1 - \beta_2^t)$ ▷ Bias corrected second moment estimate
    $\boldsymbol{\theta}_t, \boldsymbol{\phi}_t \leftarrow \{\boldsymbol{\theta}_{t-1}, \boldsymbol{\phi}_{t-1}\} - \alpha \cdot \widehat{\mathbf{m}}_t / (\sqrt{\widehat{\mathbf{v}}_t} + \epsilon)$ ▷ Update parameters
end procedure

In order to prevent overfitting during the training process, I acquired an early stopping strategy which I personally implemented. After each optimisation iteration (over the whole training set), the performance of the model was evaluated on the validation set. If the last P results were worse from the currently found best one, the training was stopped.

The whole training procedure is outlined in Algorithm 3 and the idea of early stopping is visualised in Figure 3.11.

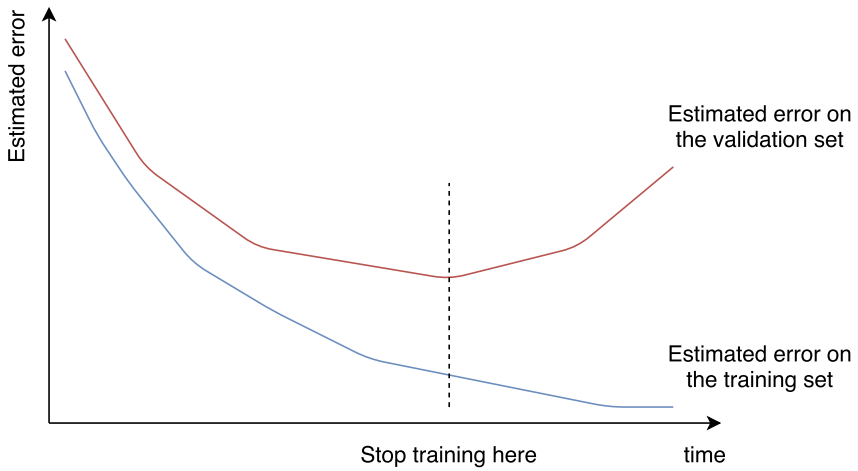


Figure 3.11: A common training scenario in which the estimated error on the validation and the training set initially decreases until overfitting begins to occur and the training process should be stopped at this point. The parameters $\boldsymbol{\theta}_t$ and $\boldsymbol{\phi}_t$ achieving the best performance on the validation set are recorded and the trained model is saved.

Algorithm 3 Training autoencoders with Adam Optimiser and early stopping. The constant value of the batch size, used for the training on ShapesSet was 100.

Require: P – early stopping patience
Require: BATCH_SIZE – minibatch size
procedure TrainAndLogAutoencoder(*autoencoder*, *train*, *validation*)
    Initialise the model parameters, $\boldsymbol{\theta}_0$ and $\boldsymbol{\phi}_0$, randomly
    $best\_score \leftarrow +\infty$, $counter \leftarrow 0$ ▷ Required for early stopping support
    while True do
        $train \leftarrow$ RandomShuffle($train$) ▷ Shuffle the training set randomly
        for $i \leftarrow 0$ **to** $|train|$ / BATCH_SIZE **do** ▷ and split it in minibatches
            PartialFit($train[i * \text{BATCH\_SIZE} : (i+1) * \text{BATCH\_SIZE}]$)
        end for
        $current\_score \leftarrow \mathcal{H}(\boldsymbol{\theta}_t, \boldsymbol{\phi}_t; valiadtion)$ ▷ Evaluate model on validation set
        if $best\_score > current\_score$ **then** ▷ Found more optimal parameters
            Save current parameters, $\boldsymbol{\theta}_t$ and $\boldsymbol{\phi}_t$, as the optimal ones
            $best\_score \leftarrow current\_score$
            $counter \leftarrow 0$
        else
            $counter \leftarrow counter + 1$
            if $counter > P$ **then** ▷ If exceeded the early stopping patience
                break ▷ stop the training to avoid overfitting
            end if
        end if
    end while
end procedure

3.2.5 Regularisation

The biggest challenge I came across during the training process was to keep the model parameters numerically stable between the iterations – an issue often arising in optimisation problems. This section describes the regularisations and the precautions taken to resolve it.

3.2.5.1 Xavier and He Initialisation

The solution to a non-convex optimisation problem, found by a gradient descent algorithm certainly depends on the initial values of its parameters, as they may sway the model towards local optima of varying quality. In the context of neural networks, it is particularly favourable for their weights to be initially distributed such that any expected input signals may usefully reach the deeper layers, as well as usefully backpropagate in the opposite direction.

The neural network weights initialisation has been found historically to be of a particular importance to the successful performance of the trained models [12, 16]. I will consider the logistic and tanh activation functions first (Figure 3.12).

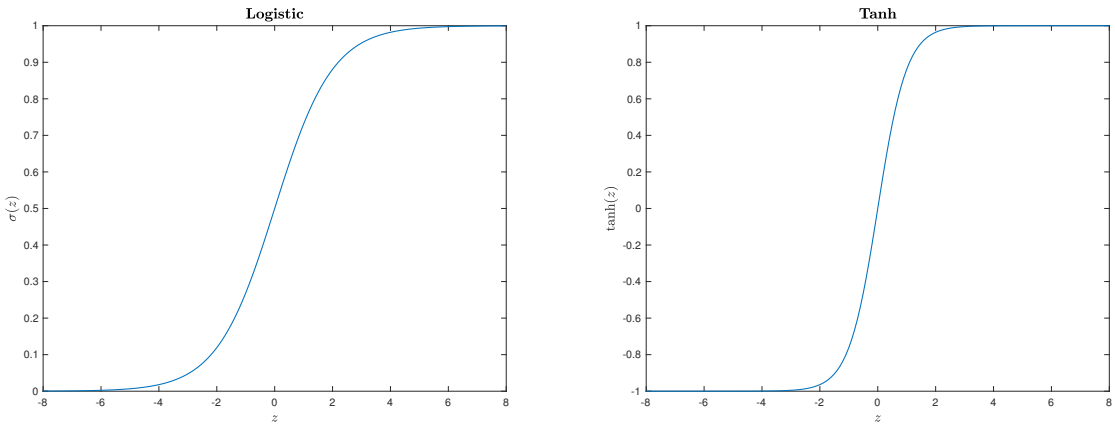


Figure 3.12: Logistic and tanh activation functions revisited.

If the neural network weights are too small, the variance of the activations across layers will decrease and the input signal will shrink when going deeper in the network. Looking at the function graphs (Figure 3.12), it can be observed that they are approximately linear around 0, hence their non-linearity effect in practice is lost and their learning capacity – greatly reduced. On the contrary, if the weights are too big, the activations will saturate as their derivatives approach 0.

Xavier initialisation [12] aims to set the weights be neither too small, nor too large in the beginning, trying to ensure none of the two problems described above occurs. The rationale behind this initialisation technique is to keep the variance of the activations the same for all layers.

Assume a set of n input values $x_1, \ldots, x_n$ with variance $\mathrm{Var}(X)$ and weights $w_1, \ldots, w_n$ with variance $\mathrm{Var}(W)$ produce an output $y = x_1 w_1 + \cdots + x_n w_n$. Then

$$\begin{aligned}\mathrm{Var}(Y) &= \mathrm{Var}\left(\sum_{i=1}^{n} x_i w_i\right) \\ &= \sum_{i=1}^{n} \mathrm{Var}(x_i w_i) \\ &= \sum_{i=1}^{n} E(X)^2 \mathrm{Var}(W) + E(W)^2 \mathrm{Var}(X) + \mathrm{Var}(X)\mathrm{Var}(W)\end{aligned} \tag{3.13}$$

Assume the inputs and the weights have zero mean: $E(X) = 0, E(W) = 0$. Then

$$\mathrm{Var}(Y) = n\mathrm{Var}(X)\mathrm{Var}(W) \tag{3.14}$$

In order to obtain $\mathrm{Var}(X) = \mathrm{Var}(Y)$, $\mathrm{Var}(W)$ should satisfy

$$\mathrm{Var}(W) = \frac{1}{n_{in}} \tag{3.15}$$

where n_{in} is the dimension of the input layer. Following the same logic for the backpropagation process, we would like

$$\mathrm{Var}(W) = \frac{1}{n_{out}} \tag{3.16}$$

to be true as well. Equations (3.15) and (3.16) can be simultaneously satisfied only for $n_{in} = n_{out}$ so Glorot and Bengio [12] consider the average metric $n_{avg} = \frac{n_{in}+n_{out}}{2}$ and establish

$$\boxed{\mathrm{Var}(W) = \frac{1}{n_{avg}} = \frac{2}{n_{in} + n_{out}}} \tag{3.17}$$

The distribution for W recommended in the literature is uniform, so $W \sim U[-a, a]$. In this case $\mathrm{Var}(W) = \frac{a^2}{3}$, so combining this standard result with Equation (3.17), we end up with

$$\textbf{Xavier initialisation:} \quad W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right] \tag{3.18}$$

This is the initialisation used for layers with logistic or tanh activation functions. The case of ReLU activations (Figure 3.13) is slightly different. The problem here is that half of the activation function's input space is effectively unused. Using a similar reasoning as already described in this section, He *et al.* [16] derived the following condition that empirically works the best for ReLU activations

$$\boxed{\mathrm{Var}(W) = \frac{2}{n_{in}}} \tag{3.19}$$

Setting W to have a zero mean Gaussian distribution this time, Equation (3.19) leads to

$$\textbf{He initialisation:} \qquad W \sim \mathcal{N}\left(0, \frac{2}{n_{in}}\right) \tag{3.20}$$

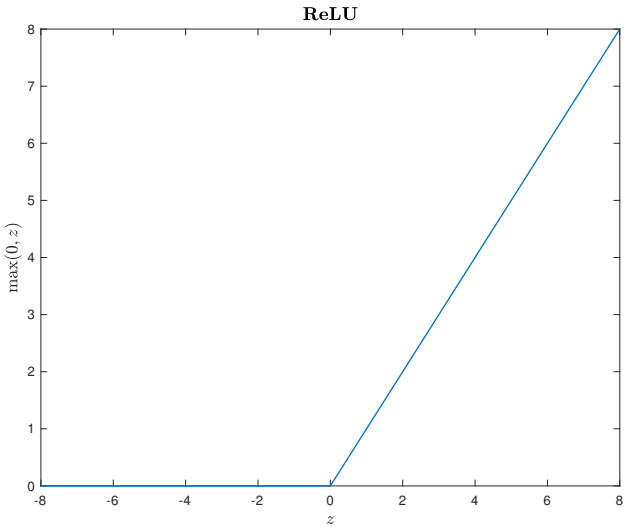


Figure 3.13: ReLU activation function revisited.

3.2.5.2 Batch Normalisation

Xavier initialisation was motivated by the desire to keep the activations' variance approximately the same across all layers. However, in general it is still possible for the distributions of layers in different depths of the network to alter, especially as its parameters change over time during the training. This is called the internal covariance shift, introduced by Ioffe and Szegedy [22] and is problematic because small perturbations in the input layers are magnified deeper in the network, making the parameters there more difficult to adapt to such changes. As a result, the network becomes harder to train effectively.

Ioffe and Szegedy [22] have proposed a method, based on batch normalisation. It begins with normalising the layers' activation to have zero mean and unit variance.

$$\widehat{x} = \frac{x - E(x)}{\sqrt{\mathrm{Var}(x)}} \tag{3.21}$$

However, performing solely this transformation may reduce the learning capacity of the network. In particular, normalising the inputs of logistic or tanh functions pushes them towards the linear region of these functions, as discussed above, counteracting any (potentially) useful saturation effects. To deal with this issue, the inserted transformation is desired to be able to reproduce the identity function, if deemed necessary. Additional learnable parameters, γ and β, are introduced for each activation which scale and shift the normalised value.

$$y = \gamma \left(\frac{x - E(x)}{\sqrt{\mathrm{Var}(x)}} \right) + \beta \tag{3.22}$$

The mean and the variance in Equation (3.22) are calculated over the whole minibatch (Figure 3.14). The exact implementation of the batch normalisation transform is given in Algorithm 4.

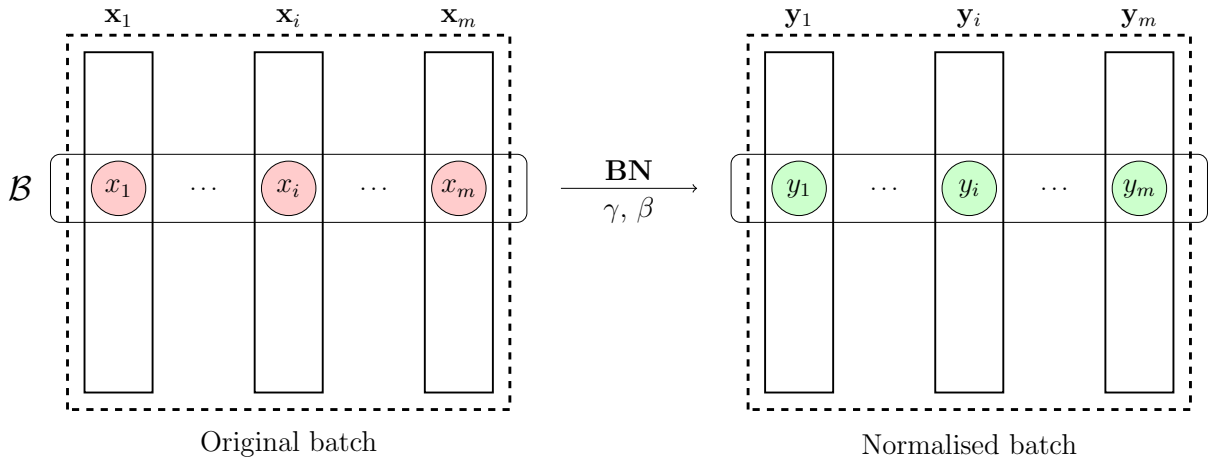


Figure 3.14: Batch normalisation transformation. Batches combine m vectors $\mathbf{x}_1, \ldots, \mathbf{x}_m$ in one group. For simplicity, the figure ignores the positional indices of the vectors elements, but it should be clear that the inputs and outputs of batch normalisation transformation correspond to the same positions.

Algorithm 4 Batch normalisation of neurons in the same position of different vectors part of the same minibatch $\mathcal{B}$. They all share the same learnable parameters γ and β, defining the scale and shift respectively. A small nonnegative constant, ϵ, is added in step 3 in order to avoid division by 0.

Input: Values of x over a minibatch: $\mathcal{B} = \{x_{1\ldots m}\}$. Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

1: $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ ▷ Minibatch mean
2: $\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ ▷ Minibatch variance
3: $\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$ ▷ Normalise
4: $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$ ▷ Scale and shift

Batch normalisation is performed before the neurons are passed through the activation functions. The updated operations pipeline is presented in Figure 3.15. The newly added learnable parameter β absorbs the bias and makes it redundant.

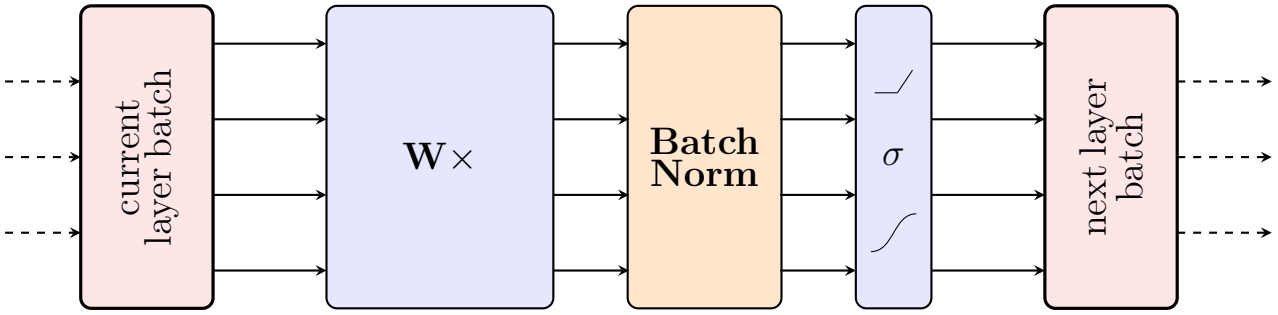


Figure 3.15: Neural network operations pipeline with batch normalisation.

The final detail is that once the neural network architecture is extended to support batch normalisation and the model is trained, the mean and the variance from Equation

(3.22) should be computed over the entire training dataset rather than a single batch when inference is performed. A possible alternative, which I used, is to keep running means and variances of the data and update them after processing each batch (similarly to what Adam Optimiser does). This allowed me to test the model performance before the training is completed.

3.3 Summary

This chapter described the implementation of the Data Management Module and the generation of my synthetic dataset, ShapesSet. An important assumption about the data was made. Precisely, some transform continuities were required so that the generating factors can be distinguished in an unsupervised approach.

Next, the Disentangled Autoencoder Suite was covered. A brief introduction to TensorFlow was given and the disentangled autoencoder construction was explained in thorough detail, emphasising all assumptions that were made in order to evaluate the loss function defined in the Preparation. The neural network was trained using the Adam optimiser with the data being fed in minibatches. The challenges, concerned with the numerical stability during the training process were resolved by introducing more suitable weights initialisation schemes (Xavier & He) and performing batch normalisation to reduce the internal covariance shift.

In the next chapter I will continue with describing the evaluation setup and will present the experimental results, produced by my models, demonstrating clear achievement and extension of my success criteria.

Chapter 4

Evaluation

This section revisits the success criteria, defined in the Project Proposal, clearly demonstrating that all of them were successfully completed or even exceeded. The evaluation setup and details of the executed experiments are described next. The correctness of the core software is established against known results from literature and it is then used for producing novel experimental results as part of the project extensions.

The evaluation was executed on ShapesSet and MNIST datasets. The exact neural network architectures (concerning layers dimensionality and activation functions) can be found in Appendix B.

4.1 Success Criteria

Referring back to the Project Proposal, the following core project success criteria were initially defined:

- *Disentangled autoencoder must be implemented and tested against test cases similar to those provided in relevant literature...*

  A detailed description of the disentangled autoencoder construction was given in Section 3.2. This criterion has been exceeded by not only implementing a fully connected autoencoder, but a convolutional one as well. It will be demonstrated that the trained models match results from relevant literature in Section 4.3.

- *A flexible test-bed should be implemented in support of the project evaluation.*

  A high level overview of the experimentation and evaluation test-bed is outlined in Section 4.2. Its flexibility is supported by the amount of results, presented in Sections 4.3, 4.4, and 4.5.

- *The effects of the autoencoder cost function coefficient on the disentanglement level must be evaluated.*

  This is achieved in Section 4.4.

All of the project extensions (*"... the disentangled features ... evaluated on classification tasks; Basic and denoising autoencoders..."*) were completed as well, allowing me to classify the project as highly successful.

4.2 Experimentation and Evaluation Setup. Integration Tests

The autoencoder construction and training, outlined in section 3.2, were at the core of the Experiment Runner implementation (Algorithm 5).

Algorithm 5 The Experiment Runner, building on the pseudocodes from the Implementation chapter.

Require: Experiment id
Require: Autoencoder settings: convolutional or fully connected; simple or denoising
Require: *Betas* – set of values for β we would like to evaluate afterwards

```
train, test, validation = GenerateShapesSet(id)
for β in Betas do
    autoencoder = BuildAutoencoder(id, settings, β)
    TrainAndLogAutoencoder(autoencoder, train, validation)
end for
```

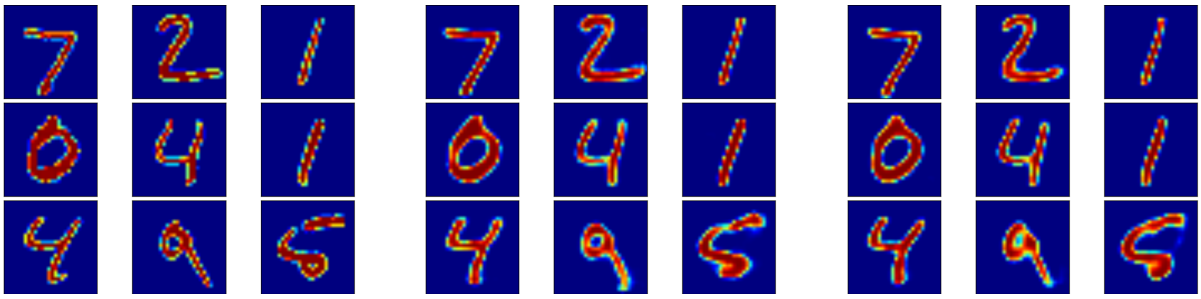
Once the autoencoders were trained and saved, they could be reloaded again and used for obtaining various evaluation metrics (Algorithm 6).

Algorithm 6 A very high-level overview of the Evaluation Suite.

```
for all trained autoencoders do
    autoencoder ← RestoreAutoencoder(id, settings, β)
    Evaluate(autoencoder, test)
end for
```

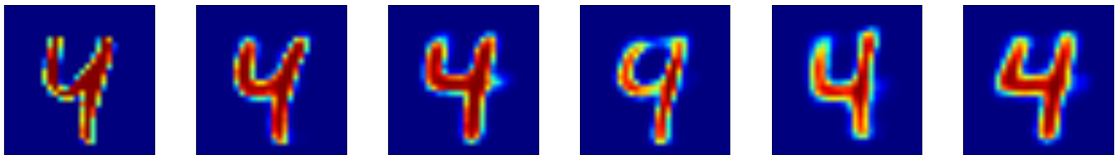
The entirety of the evaluation was undertaken on a GPU server, provided by the Computational Biology group at the Computer Laboratory. Integration tests were run locally on my computer before deploying the code to the GPU.

The test was based on the fact that the autoencoders' primary goal, before everything else, is to reconstruct the original inputs in their output layers. Some of the results, produced by the testing suite, run on both fully connected and convolutional architectures for various values of β, are presented in Figures 4.1, 4.2, and 4.3. The tests used the MNIST dataset. The training iterations of the test autoencoders were limited to 20.



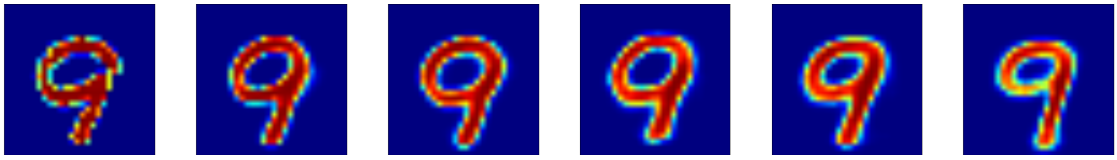
(a) Original images. (b) Fully connected, $\beta = 0$. (c) Convolutional, $\beta = 0$.

Figure 4.1: Autoencoders, trained with $\beta = 0$. The loss function forces as accurate reconstructions as possible, ignoring the KL-divergence term. (a) includes original MNIST images, while (b) and (c) are reconstructions, generated by the autoencoders.



(a) original (b) $\beta = 1$ (c) $\beta = 2$ (d) $\beta = 3$ (e) $\beta = 4$ (f) $\beta = 5$

Figure 4.2: Fully connected autoencoder reconstructions for different values of β.



(a) original (b) $\beta = 1$ (c) $\beta = 2$ (d) $\beta = 3$ (e) $\beta = 4$ (f) $\beta = 5$

Figure 4.3: Convolutional autoencoder reconstructions for different values of β.

As can be intuitively observed in Figures 4.2 and 4.3, with increasing the values for β, the autoencoder reconstructions alter more and more from the original image, but some distinct features of the digits become "stronger" and the outputs seem more generalised.

In the next section I will show that my models achieve the desired disentanglement effect when trained on ShapesSet for values of β, cited in literature.

4.3 Demonstrating disentanglement on ShapesSet

In order to complete my core project successfully, I had to evaluate the behaviour of my trained autoencoders against known results from literature. Higgins *et al.* [17] claimed that autoencoders, trained with $\beta = 4$, achieve a good level of disentanglement when applied to the shapes dataset they used for testing. In this section I reproduce this result, evaluating the models on the previously described ShapesSet.

The code dimension of the autoencoders trained on ShapesSet was set to 10 (Appendix B), following the neural network architectures prescribed in literature. I will first consider

the mean of each code element with respect to each generating factor. That is, consider element i of the code (i.e. the latent z_i). We fix the value for some generating factor (say f') and iterate over the values of all other factors ($\boldsymbol{f}$) such that $img = GenerateImage(f' \cup \boldsymbol{f}) \in test$ (i.e. in the evaluation process only images from the test subset, unobserved during the training, were considered). The *GenerateImage* function here is the same as the one used in Algorithm 1. For each such img, $\mathbf{z}^{\mu} = encoder(img)$, the learnt code means (the $\boldsymbol{\mu}$ layer in Figure 3.9) are generated, the latent z_i^{μ} is taken and in the end all such values are averaged. Note that, the learnt means of the code are extracted, rather than the sampled codes themselves for better statistical precision. Formally, this is precisely

$$\begin{aligned} latent\_mean\_value(z^{\mu}_{i,f'}) = average\{z_i^{\mu} | \mathbf{z}^{\mu} = encoder(img), \\ img = GenerateImage(f' \cup \boldsymbol{f}) \in train\} \end{aligned} \tag{4.1}$$

for factor f' fixed and iterating over all other generating factors $\boldsymbol{f}$.

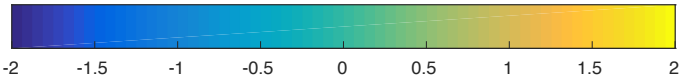
The obtained results when f' is chosen to be in turn the position of the shape, its size and rotation, are visualised in Figure 4.4.

From the presented diagrams, it can be inferred that as a side effect of the pressure put by the disentanglement parameter β, the autoencoder automatically and in an unsupervised way "delegates" the task of encoding shapes positions in the image solely to latents z_0 and z_8, whereas z_1 is fully responsible for determining their size. The rotation factor is the hardest to disentangle and has five latents devoted to it (although not all of them are active for all shapes) but it is clear they have learnt some form of wave functions with the respective rotation period for each shape. Latents z_3 and z_6 are completely switched off for all factors. The main advantages of the underlying learning representation, constructed by the autoencoder, are its interpretability and predictability. We can sometimes more easily reason about what features of the input data relate to what values of the code. Moreover, small perturbations of the code lead to expected changes of the decoder output reconstructions (Figure 4.5).

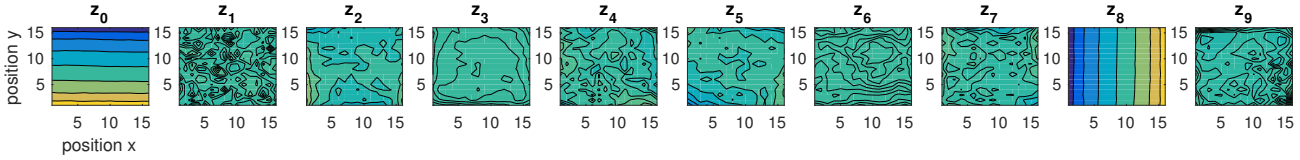
A similar analysis was undertaken for the entangled case when $\beta = 0$ (Figures 4.6 and 4.7). ***The outcomes of the executed experiments are in line with known results, cited in literature for fixed values of*** $\beta = \{0, 4\}$***, and establish the correct implementation of the core project software.*** In the next section one of the project extensions is presented and the effect of intermediate values of β to the models' behaviour is examined.

4.4 Measuring disentanglement. Investigating intermediate values of β

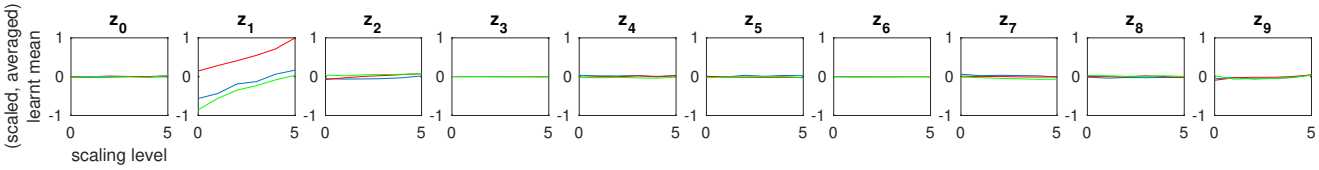
The disentanglement of an autoencoder cannot be usefully measured by its reconstruction accuracy or the KL-divergence term of the loss function as they fail to convey the notion of independence we want to obtain for the elements of the code. Precisely, disentanglement effect would mean distinguishing the generating factors of the data and encoding them in separate code elements, as partially achieved in Figure 4.4.



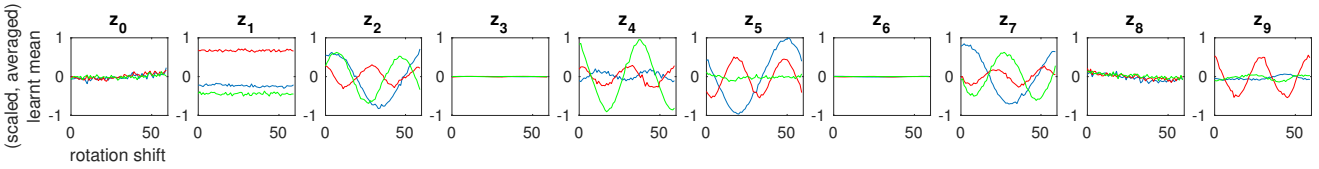
(a) Numerical range of the colours in the contour diagrams below.



(b) $f' = \{positionX, positionY\}$, $\boldsymbol{f} = \{scale, rotation\}$. Learnt means of each latent z_i as a function of all 16×16 locations, averaged across objects, rotations and scales.



(c) $f' = \{scale\}$, $\boldsymbol{f} = \{positionX, positionY, rotation\}$. Learnt means of each latent z_i as a function of all 6 scale levels, averaged across rotations and positions. The red line corresponds to the results for square shapes, the blue one for ellipses, and the green one for triangles.



(d) $f' = \{rotation\}$, $\boldsymbol{f} = \{positionX, positionY, scale\}$. Learnt means of each latent z_i as a function of all 60 rotation factor values, averaged across scales and positions. Once again, red lines present the results for square shapes, blue ones – for ellipses, and, green ones – for triangles.

Figure 4.4: Considering the mean of each element of the code when fixing a particular factor and averaging across all the other generating factors. Results are for an autoencoder trained with $\beta = 4$, achieving a good level of disentanglement. The graphs in (c) and (d) are scaled to the $[-1, +1]$ region.

Higgins *et al.* [17] propose a disentanglement measuring method which tries to evaluate this property of the trained autoencoders. A random set of generating factors is taken, the image img_1 is constructed, and the code means $\mathbf{z}_1^\mu = encoder(img_1)$ are extracted. The same procedure is repeated, but this time one of the factors is randomly modified while all the others are kept the same. Denote the newly extracted code means with $\mathbf{z}_2^\mu$. A low capacity linear classifier is trained to map $\frac{|\mathbf{z}_1^\mu - \mathbf{z}_2^\mu|}{\max(|\mathbf{z}_1^\mu - \mathbf{z}_2^\mu|)}$ (division intended for normalisation) to the single factor that was changed during the process of obtaining $\mathbf{z}_1^\mu$ and $\mathbf{z}_2^\mu$. The classifier accuracy is then reported as a disentanglement measure of the autoencoder of interest. The complete procedure, which was implemented for generating the data required for training and evaluating the classifier, is outlined in Appendix C. The assumption is that if a simple classifier is capable of inferring what small changes of the input data are responsible for the changes in the code, then the model provides some form of transparency and interpretability. Figure 4.8 presents the disentanglement levels of four types of autoencoders I trained within this project, varying β from 0 to 5 with a

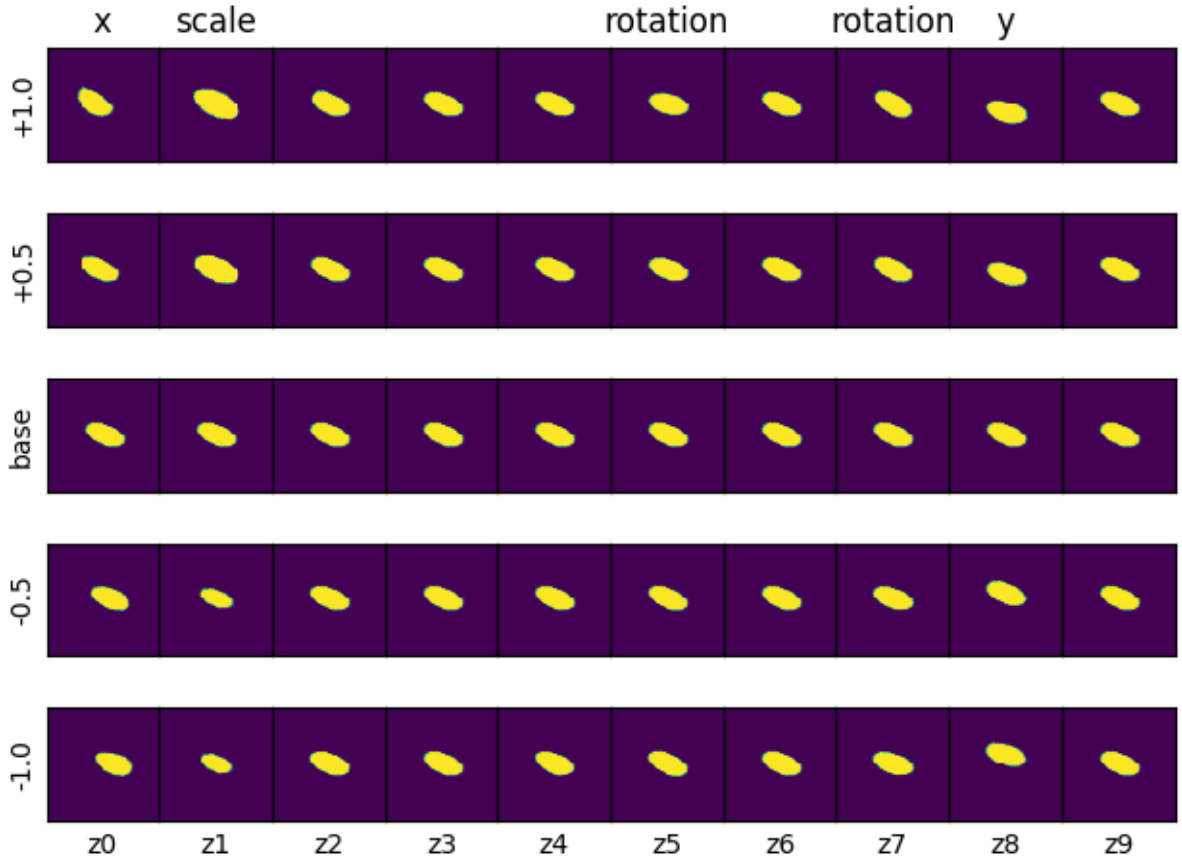
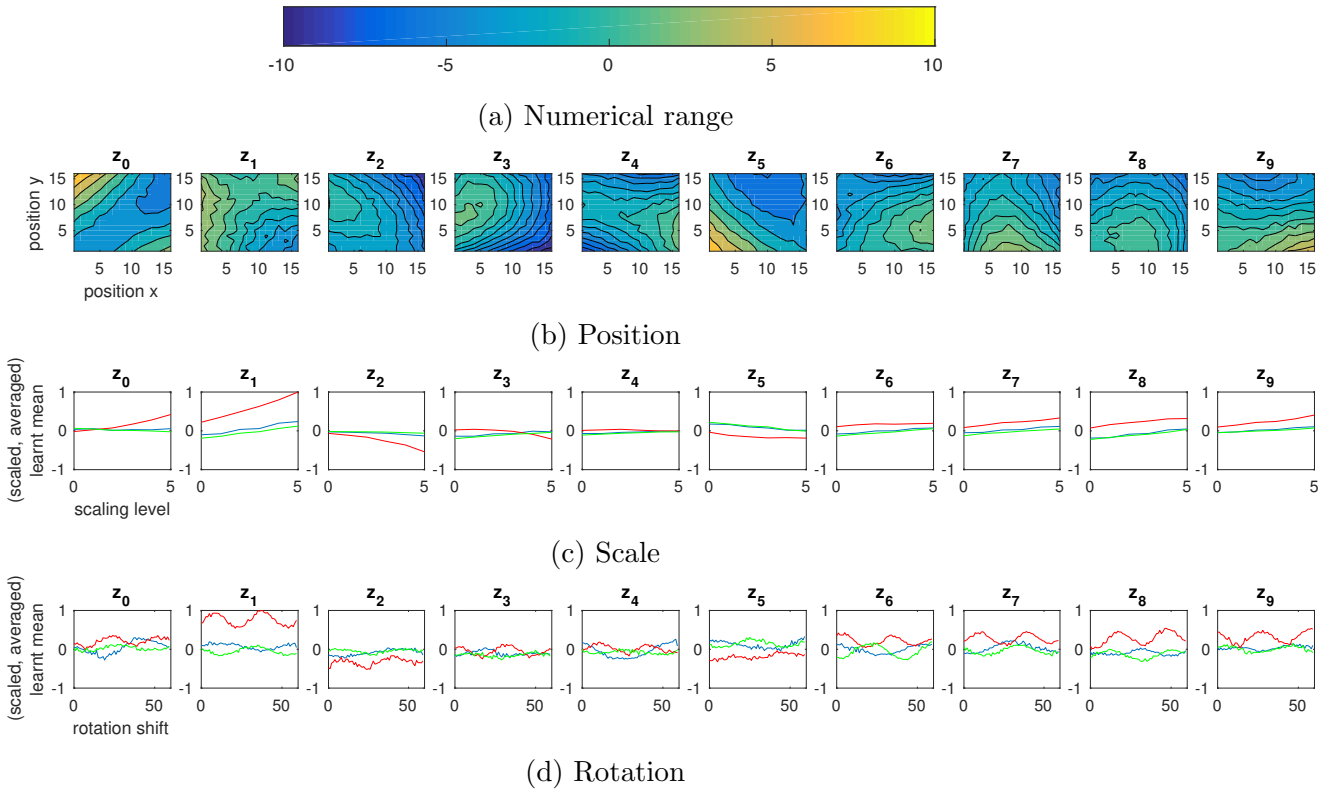


Figure 4.5: Decoder reconstructions, resulting from small variations of the code. A base image was taken and its code was generated by the encoder network. The figure presents the decoder's predictable reconstructions when a small Δ (± 0.5, ± 1) is added to each code element individually. The latent variables postulated as responsible for encoding position, scale and rotation factors, modify the output accordingly. (The results are for the disentangled case, $\beta = 4$.)

step of 0.2. For the purposes of this figure and all subsequent ones, error bars will refer to standard deviations. My implementation of the low-capacity linear classifier, used in the evaluation process, is described in Appendix D.

The first thing which becomes clear from the results is the high variance between separate runs with the same value for β. A potential reason for that might be the method not being completely capable of closing the gap between the notion of disentanglement and factor independence we have with the underlying properties of the representations learnt by the autoencoders. For example, it was observed that for the position latents in the case of $\beta = 4$, the autoencoder may sometimes learn "curved" or rotated, but still orthogonal, coordinate systems, which differs from what we would expect. Moreover, when reporting their results in [17], the bottom 50% of the obtained measurements have been discarded for unknown reasons (this was not performed when organising my results in Figure 4.8). In a subsequent paper [18], the way of computing the disentanglement metric



(a) Numerical range

(b) Position

(c) Scale

(d) Rotation

Figure 4.6: Demonstrating the entangled representation for an autoencoder, trained with $\beta = 0$. All of the code variables are active and contribute for the encoding of more than one generating factor.

was modified. Instead of randomly changing the value of one generating factor, this time one generating factor is kept constant and all the others are randomly altered. [18] was presented at a conference less than a month before the dissertation deadline and due to time constraints, this approach was not evaluated. [17] and [18] report results for fixed values of β ($\beta = 0, 1$, and 4) only, so to the best of my knowledge the findings about the intermediate values, presented in Figure 4.8 are novel and unpublished.

The second trend that is observed is the increase in the disentanglement with bigger values for β. The growth seems to be the most steady for convolutional denoising autoencoders. This is consistent with the claims that convolutional networks might be better at capturing image structures than fully connected ones and that adding noise and reconstructing the original data could act as a good regulariser.

The assumption for bigger values of β is that at some point the autoencoder disentanglement will get flat (as starting to happen for the fully connected denoising case) and from then onwards further increase of β will be damaging, as it will come at the cost of reducing the autoencoder's reconstruction ability. This in turn can lead to losing some useful learnt propertiest about the data. An application in which even small β can be harmful is described in the next section.

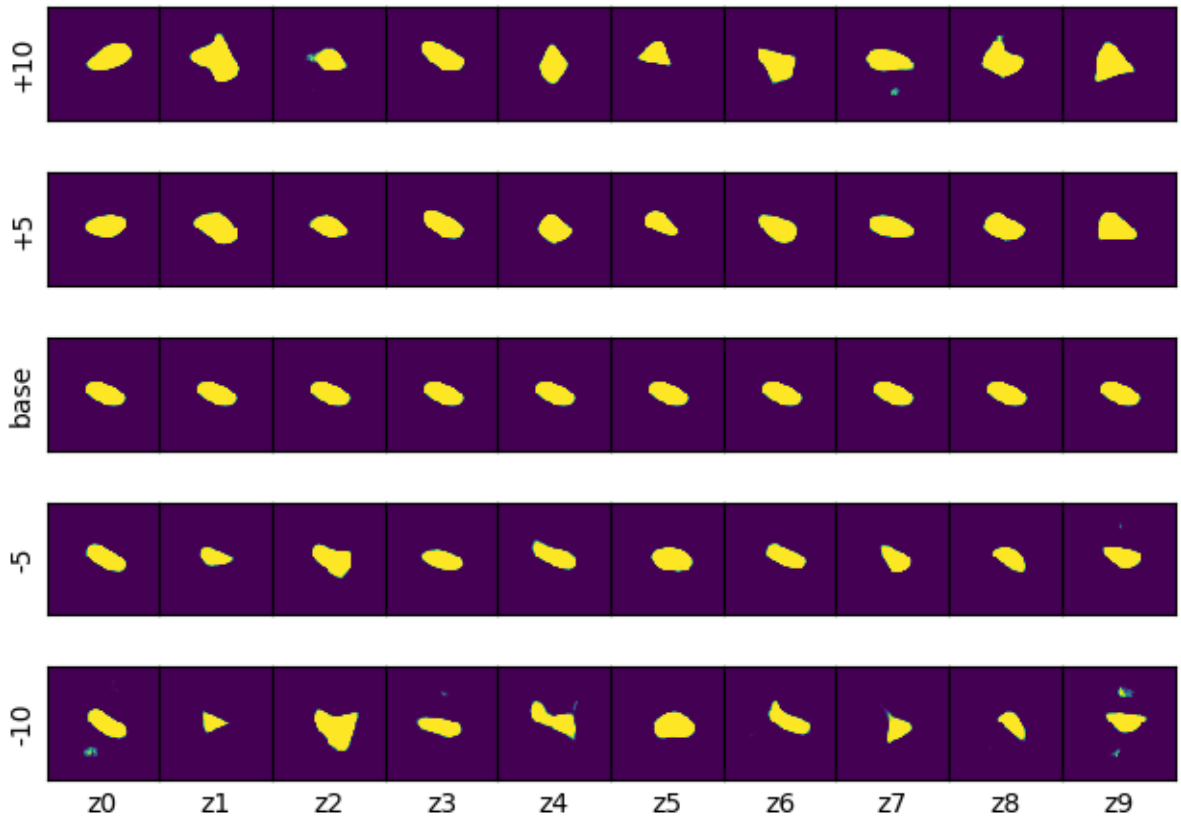


Figure 4.7: The entangled case ($\beta = 0$). Small changes in the code lead to uninterpretable reconstructions. The values of Δ, indicating the applied code disturbance, should not be considered as absolute, but rather than relative to the model, as the numerical ranges, covered by the codes for both cases (Figures 4.4a and 4.6a) are different.

4.5 MNIST classification with disentangled autoencoders

After evaluating disentangled autoencoders' behaviour on my synthetic dataset, it was a natural continuation to test them against an established machine learning benchmark. The MNIST classification problem is well studied for the past 20 years and as such was considered to be a suitable candidate. The evaluation procedure began with an unsupervised autoencoder training first. Subsequently, a Support Vector Machine[1] classifier from the `scikit-learn` library was trained to map the image codes, produced by the encoder network, to the respective image classes. The results are presented in Figure 4.9.

It is interesting to observe the major spikes in the classification accuracy from $\beta = 0$ to $\beta = 0.1$ since afterwards the accuracy is steadily dropping. Values of β between 0 and 0.1 might posses some special properties from both theoretical and implementation

[1]Covered by the Part II Machine Learning and Bayesian Inference course. Not discussed in this dissertation further.

perspective but no further studies have been made in that direction.

Taking into account the increasing error in the autoencoders reconstruction precision as well (Figure 4.10), it can be concluded that the autoencoder disentanglement is deteriorating for classification problems when applied to the MNIST dataset. This is an expected result, especially because of the lack of explicit continuity and generating factors of the MNIST images. It establishes the fact that there is a trade-off between the two terms of the disentangled autoencoder loss function and that they force the model learn different properties about the data. When training a disentangled autoencoder, this trade-off should be considered and a balanced solution is desirable.

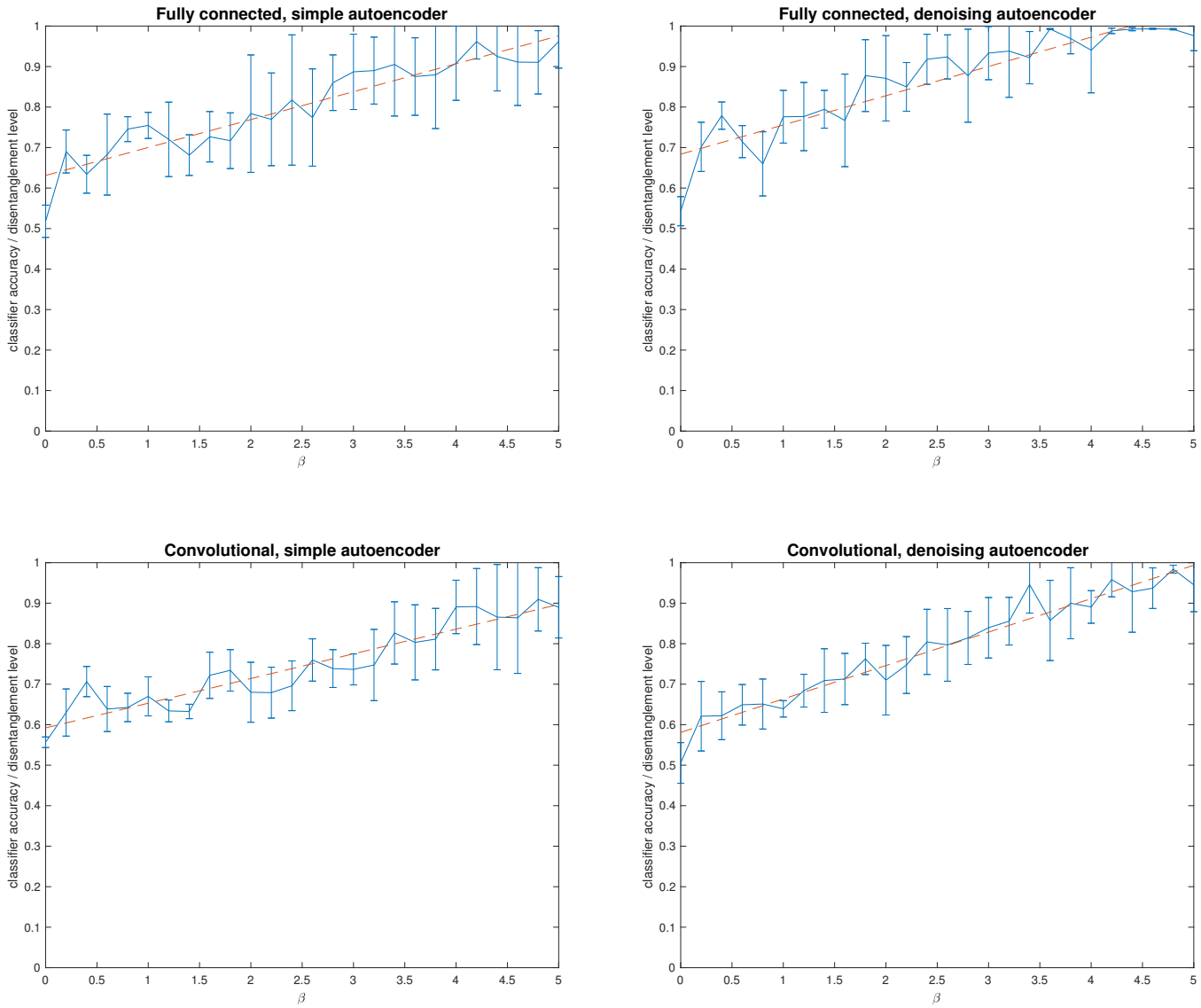


Figure 4.8: Disentanglement levels of the autoencoders, trained on ShapesSet, with respect to the parameter β. Simple and denoising variants of autoencoders were considered and both fully connected and convolutional architectures were tested. The applied noise in the denoising case was salt-and-pepper, randomly flipping up to 20% of the pixels. For each β, 5 autoencoder models were trained. In all graphs here and below I plot the means of the results obtained for all models trained with the same β while the error bars denote standard deviations. The red line indicates the trend in the autoencoder' behaviour. Their disentanglement is increasing with increasing values of β.

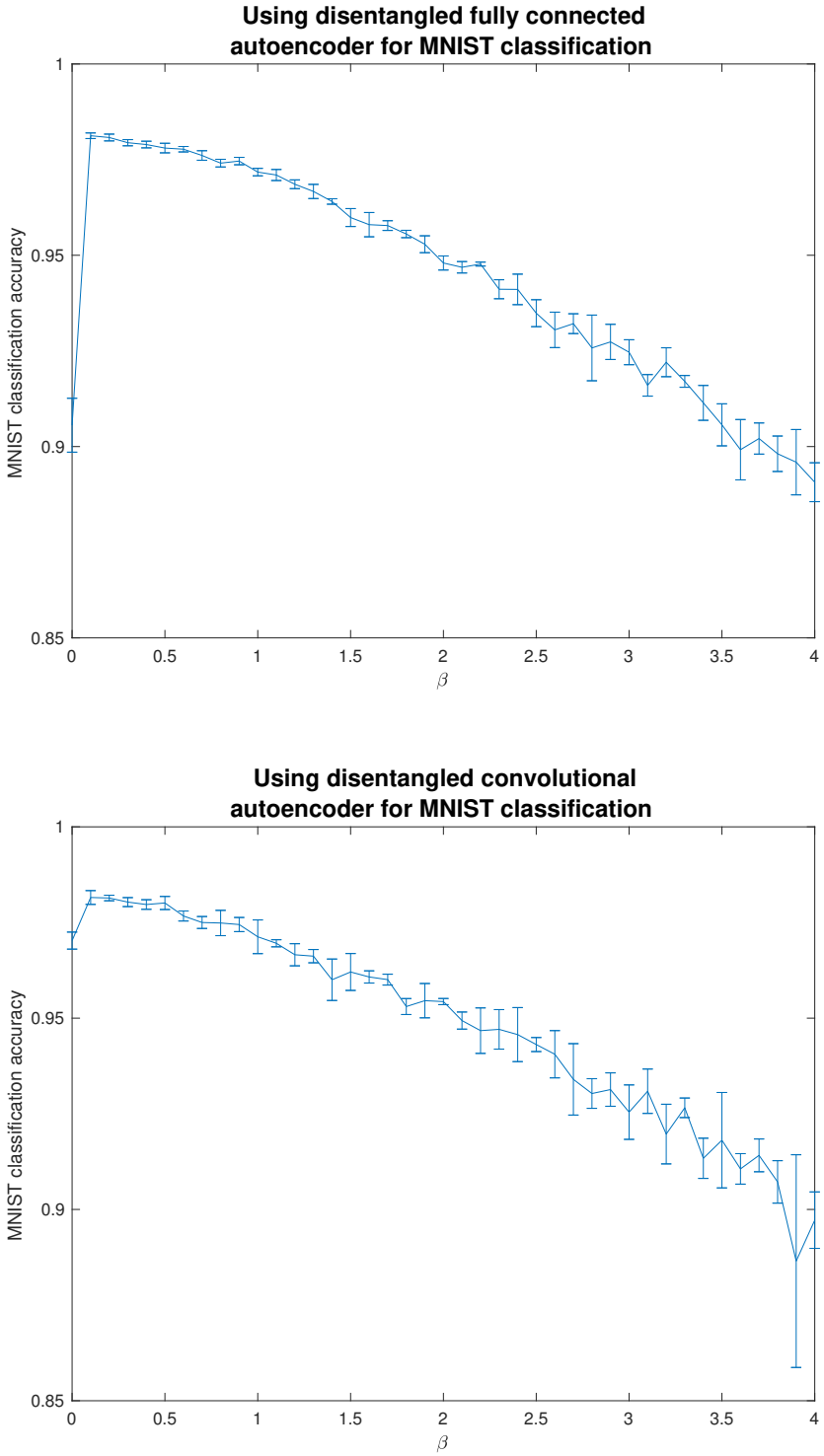


Figure 4.9: Results of MNIST classification with autoencoders. The bigger variance in the convolutional autoencoder results comes from the fact a smaller number of samples was managed to be obtained during the evaluation process. The range for β is from 0 to 4 with step 0.1.

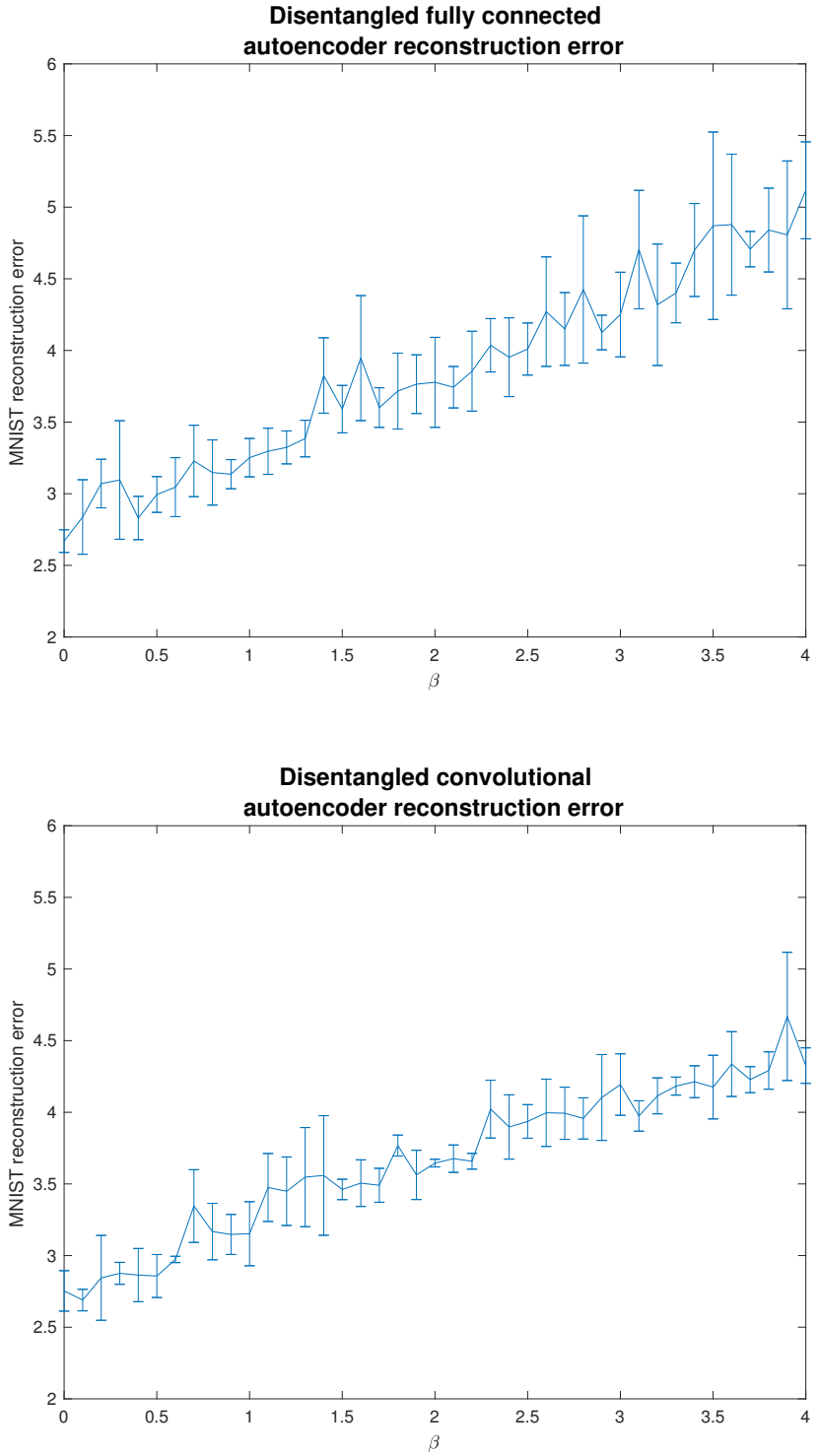


Figure 4.10: Measuring autoencoders' reconstruction error on MNIST. If $\mathbf{x}$ is the autoencoder input and $\mathbf{x}'$ is the autoencoder reconstruction, reported here is $\frac{1}{n}\|\mathbf{x} - \mathbf{x}'\|_2$, where n is the dimension of $\mathbf{x}$ and $\mathbf{x}'$. The fully connected autoencoder's reconstruction error grows faster than that of the convolutional autoencoder.

Chapter 5

Conclusions

5.1 Achievements

The project was a huge success, achieving and extending all requirements. Recent scientific results were reproduced, demonstrating the disentanglement achieved by an autoencoder, trained on the synthetic ShapesSet. Independent generating factors of the input data were written in separate places of the neural network, improving on its transparency and interpretability.

I managed to contribute with, to the best of my knowledge, new and unpublished findings about the properties of disentangled autoencoders. In particular, their level of disentanglement was measured over a whole range of values β and it was discovered that, as expected, the disentanglement typically makes the models' performance worse in classification tasks.

5.2 Lessons Learnt

Apart from all of the extensive theoretical knowledge I had to familiarise myself with, the most important lesson I will take away from this project is to look for ways of shortening the time cycle between training and evaluating machine learning models such that the robustness and the stability of the software are preserved. There were moments in which the training of models with inappropriate settings could potentially be detected earlier saving precious time spent on waiting for the training to complete.

5.3 Further Work

Although the project is successful and completed, the results obtained in the Evaluation chapter pose some interesting questions which are worth examining further:

- *Huge variance between different runs with the same value for β.* Although there exist general trends in the models behaviour, it is not clear what led to the big discrepancies in the experimental results, received by autoencoders trained with the same β. Potential reasons for that might be glitches in the training process, the

neural network architecture being too restrictive, or this might be simply an inherent property of those models.

- *Alternative Disentnaglement Measurement.* The above issue might be tackled by proposing a more suitable method for measuring disentanglement than the one implemented in this project. An open question is if it can be theoretically as well as empirically justified.

- *More complex datasets.* It is yet to be established if the method concerned in this work may provide any advantages when applied to more complex datasets rather than the basic ShapesSet for example.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Guillaume Alain and Yoshua Bengio. What regularized auto-encoders learn from the data-generating distribution. *J. Mach. Learn. Res.*, 15(1):3563–3593, January 2014.

[3] Yoshua Bengio, Li Yao, Guillaume Alain, and Pascal Vincent. Generalized denoising auto-encoders as generative models. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 899–907. Curran Associates, Inc., 2013.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[5] H. Bourlard and Y. Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, 59(4):291–294, 1988.

[6] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[7] Brian Cheung, Jesse A. Livezey, Arjun K. Bansal, and Bruno A. Olshausen. Discovering hidden factors of variation in deep networks. *CoRR*, abs/1412.6583, 2014.

[8] François Chollet. Keras. `https://github.com/fchollet/keras`, 2015.

[9] Taco S. Cohen and Max Welling. Transformation properties of learned visual representations. *CoRR*, abs/1412.7659, 2014.

[10] G. Desjardins, A. Courville, and Y. Bengio. Disentangling Factors of Variation via Generative Entangling. *arXiv*, October 2012.

[11] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *ArXiv e-prints*, March 2016.

[12] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, 2010.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] Ross Goroshin, Michael F Mathieu, and Yann LeCun. Learning to linearize under uncertainty. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1234–1242. Curran Associates, Inc., 2015.

[15] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 1026–1034, Washington, DC, USA, 2015. IEEE Computer Society.

[17] Irina Higgins, Loïc Matthey, Xavier Glorot, Arka Pal, Benigno Uria, Charles Blundell, Shakir Mohamed, and Alexander Lerchner. Early visual concept learning with unsupervised deep learning. *CoRR*, abs/1606.05579, 2016.

[18] Irina Higgins, Loïc Matthey, Xavier Glorot, Arka Pal, Benigno Uria, Charles Blundell, Shakir Mohamed, and Alexander Lerchner. β-VAE: Learning basic visual concepts with a constrained variational framework. *ICLR*, 2017.

[19] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov 2012.

[20] Geoffrey E. Hinton, Alex Krizhevsky, and Sida D. Wang. Transforming auto-encoders. In *Proceedings of the 21th International Conference on Artificial Neural Networks - Volume Part I*, ICANN'11, pages 44–51, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] Geoffrey E Hinton and Richard S. Zemel. Autoencoders, minimum description length and helmholtz free energy. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 3–10. Morgan-Kaufmann, 1994.

[22] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[23] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[24] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2013.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[27] Matej Moravcík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael H. Bowling. Deepstack: Expert-level artificial intelligence in no-limit poker. *CoRR*, abs/1701.01724, 2017.

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[29] Scott Reed, Kihyuk Sohn, Yuting Zhang, and Honglak Lee. Learning to disentangle factors of variation with manifold interaction. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages II–1431–II–1439. JMLR.org, 2014.

[30] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. *CoRR*, abs/1503.03832, 2015.

[31] Shaked Shammah Shai Shalev-Shwartz, Ohad Shamir. Failures of deep learning. *arXiv preprint arXiv:1703.07950*, 2017.

[32] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 01 2016.

[33] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013.

[34] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, June 2014.

[35] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016.

[36] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1096–1103, New York, NY, USA, 2008. ACM.

[37] Zhenyao Zhu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning multi-view representation for face recognition. *CoRR*, abs/1406.6947, 2014.

Appendix A

Further theory

A.1 Backpropagation Equation Derivations

In the main text we defined

$$z_j^{l+1} = \sum_{\substack{1 \le i \le n_l \\ w_{i\to j}^l \neq \perp}} w_{i\to j}^l x_i^l + b_j^{l+1} \tag{A.1}$$

$$x_i^l = \sigma(z_i^l) \tag{A.2}$$

$$\delta_i^l = \frac{\partial \mathcal{L}}{\partial z_i^l} \tag{A.3}$$

Equation (2.12):

$$\boxed{\delta_i^M = \frac{\partial \mathcal{L}}{\partial x_i^M} \cdot \sigma'(z_i^M)}$$

$$\begin{aligned}
\delta_i^M &= \frac{\partial \mathcal{L}}{\partial z_i^M} && \text{By definition (A.3)} \\
&= \frac{\partial \mathcal{L}}{\partial x_i^M} \frac{\partial x_i^M}{\partial z_i^M} && \text{Chain rule} \\
&= \frac{\partial \mathcal{L}}{\partial x_i^M} \frac{\partial \sigma(z_i^M)}{\partial z_i^M} && \text{By definition (A.2)} \\
&= \frac{\partial \mathcal{L}}{\partial x_i^M} \cdot \sigma'(z_i^M) && \text{As required.}
\end{aligned}$$

Equation (2.13):

$$\boxed{\delta_i^l = \left(\sum_{\substack{1 \le j \le n_{l+1} \\ w_{i\to j}^l \neq \perp}} w_{i\to j}^l \delta_j^{l+1} \right) \sigma'(z_i^l)}$$

$$\begin{aligned}
\delta_i^l &= \frac{\partial \mathcal{L}}{\partial z_i^l} && \text{By definition (A.3)} \\
&= \sum_{\substack{1 \le j \le n_{l+1} \\ w_{i\to j}^l \neq \perp}} \frac{\partial \mathcal{L}}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_i^l} && \text{Chain rule} \\
&= \sum_{\substack{1 \le j \le n_{l+1} \\ w_{i\to j}^l \neq \perp}} \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial z_i^l} && \text{By definition (A.3)} \\
&= \sum_{\substack{1 \le j \le n_{l+1} \\ w_{i\to j}^l \neq \perp}} \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial x_i^l} \frac{\partial x_i^l}{\partial z_i^l} && \text{Chain rule} \\
&= \left(\sum_{\substack{1 \le j \le n_{l+1} \\ w_{i\to j}^l \neq \perp}} \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial x_i^l} \right) \sigma'(z_i^l) && \text{From (A.2)} \\
&= \left(\sum_{\substack{1 \le j \le n_{l+1} \\ w_{i\to j}^l \neq \perp}} \delta_j^{l+1} \frac{\partial}{\partial x_i^l} \left(\sum_{\substack{1 \le k \le n_l \\ w_{k\to j}^l \neq \perp}} w_{k\to j}^l x_k^l + b_j^{l+1} \right) \right) \sigma'(z_i^l) && \text{By definition (A.1)} \\
&= \left(\sum_{\substack{1 \le j \le n_{l+1} \\ w_{i\to j}^l \neq \perp}} \delta_j^{l+1} w_{i\to j}^l \right) \sigma'(z_i^l) && \text{As required.}
\end{aligned}$$

Equation (2.14):

$$\boxed{\frac{\partial \mathcal{L}}{\partial w_{i\to j}^l} = \delta_j^{l+1} x_i^l}$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_{i\to j}^l} &= \frac{\partial \mathcal{L}}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial w_{i\to j}^l} && \text{Chain rule} \\
&= \delta_j^{l+1} \frac{\partial}{\partial w_{i\to j}^l} \left(\sum_{\substack{1 \le k \le n_l \\ w_{k\to j}^l \neq \perp}} w_{k\to j}^l x_k^l + b_j^{l+1} \right) && \text{By definitions (A.3) and (A.1)} \\
&= \delta_j^{l+1} x_i^l && \text{As required.}
\end{aligned}$$

Equation (2.15):

$$\boxed{\frac{\partial \mathcal{L}}{\partial b_i^l} = \delta_i^l}$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial b_i^l} &= \frac{\partial \mathcal{L}}{\partial z_i^l}\frac{\partial z_i^l}{\partial b_i^l} && \text{Chain rule} \\
&= \delta_i^l \frac{\partial}{\partial b_i^l}\left(\sum_{\substack{1\le k\le n_{l-1} \\ w_{k\to i}^{l-1}\neq \perp}} w_{k\to i}^{l-1} x_k^{l-1} + b_i^l\right) && \text{By definitions (A.3) and (A.1)} \\
&= \delta_i^l && \text{As required.}
\end{aligned}$$

A.2 Activation Function Derivatives

- Identity: $\sigma(x) = x$

$$\sigma'(x) = 1$$

- Rectified linear unit (ReLU): $\sigma(x) = \max(0, x)$

$$\sigma'(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \\ \perp & x = 0 \end{cases}$$

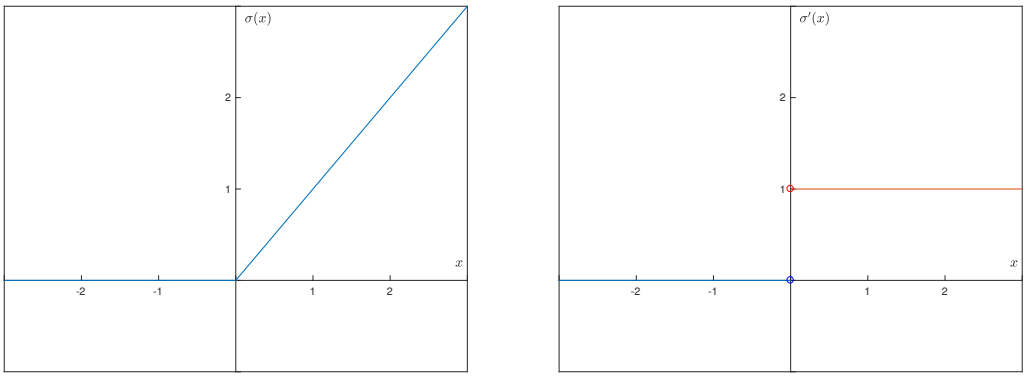


Figure A.1: The ReLU function on the left and its derivative on the right.

- Logistic sigmoid: $\sigma(x) = \frac{1}{1+\exp(-x)}$

$$\begin{aligned}
\sigma'(x) &= \left[\frac{1}{1+\exp(-x)}\right]' = \left[(1+\exp(x))^{-1}\right]' \\
&= -\frac{1}{(1+\exp(-x))^2}\,(1+\exp(-x))' \\
&= \sigma(x)\frac{\exp(-x)}{1+\exp(-x)} \\
&= \sigma(x)\frac{1+\exp(-x)-1}{1+\exp(-x)} \\
&= \sigma(x)\left[1-\frac{1}{1+\exp(-x)}\right] \\
&= \sigma(x)\left[1-\sigma(x)\right]
\end{aligned}$$

- Tanh: $\sigma(x) = \tanh(x)$

$$\sigma'(x) = 1 - \tanh^2(x)$$

A.3 Kullback–Leibler Divergence

If $Q(x)$ and $P(x)$ are two probability distributions defined over the same set of outcomes x, then their Kullback–Leibler divergence is defined to be the integral

$$D_{KL}(Q\|P) = \int_{-\infty}^{\infty} q(x)\log\frac{q(x)}{p(x)}\,\mathrm{d}x \tag{A.4}$$

and is a measure of the information gain by approximating the distribution P with Q. $D_{KL}(Q\|P) \geq 0$ and the equality is achieved when $Q = P$ (intuitively, in this case the distance between the two distributions is 0). The Kullback–Leibler divergence is not, however, a true metric for distance because in general it lacks symmetry: $D_{KL}(Q\|P) \neq D_{KL}(P\|Q)$. It does not obey the triangle inequality neither.

A.4 Evidence Lower Bound (ELBO) – Derivations

We defined the evidence lower bound (ELBO) to be

$$\text{ELBO}(\theta, \phi) = \log p_\theta(\mathbf{x}) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}|\mathbf{x})) \tag{A.5}$$

However, we can write

$$\begin{aligned}
\log p_\theta(\mathbf{x}) &= 1. \log p_\theta(\mathbf{x}) \\
&= \left(\int_Z q_\phi(\mathbf{z}|\mathbf{x}) \, \mathrm{d}\mathbf{z} \right) \log p_\theta(\mathbf{x}) \\
&= \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \log p_\theta(\mathbf{x}) \, \mathrm{d}\mathbf{z} \\
&= \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{p_\theta(\mathbf{z}|\mathbf{x})} \, \mathrm{d}\mathbf{z} \\
&= \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{z}, \mathbf{x}) q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z}|\mathbf{x}) q_\phi(\mathbf{z}|\mathbf{x})} \, \mathrm{d}\mathbf{z} \\
&= \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \log \left(\frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})} \frac{p_\theta(\mathbf{z}|\mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})} \right) \mathrm{d}\mathbf{z} \\
&= \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \left(\log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})} + \log \frac{p_\theta(\mathbf{z}|\mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})} \right) \mathrm{d}\mathbf{z} \\
&= \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})} \, \mathrm{d}\mathbf{z} + \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{z}|\mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})} \, \mathrm{d}\mathbf{z} \\
&= \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})} \, \mathrm{d}\mathbf{z} + D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}|\mathbf{x}))
\end{aligned}$$

From (A.5) it now follows that

$$\begin{aligned}
\text{ELBO}(\theta, \phi) &= \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{z}, \mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})} \, \mathrm{d}\mathbf{z} \\
&= \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{x}|\mathbf{z}) p_\theta(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \, \mathrm{d}\mathbf{z} \\
&= \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \left(\log p_\theta(\mathbf{x}|\mathbf{z}) + \log \frac{p_\theta(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \right) \mathrm{d}\mathbf{z} \\
&= \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \log p_\theta(\mathbf{x}|\mathbf{z}) \, \mathrm{d}\mathbf{z} - \int_Z q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z})} \, \mathrm{d}\mathbf{z} \\
&= \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}))
\end{aligned}$$

A.5 Disentangled Autoencoder Loss Function Alternative Derivation

This approach closely follows the paper by Higgins *et al.* [17].

Again, assume an approximation $q_\phi(\mathbf{z}|\mathbf{x})$ of the intractable $p_\theta(\mathbf{z}|\mathbf{x})$. Maximum likelihood training of the autoencoder aims to maximise the probability of observing the data averaged over all latent values for $\mathbf{z}$, precisely:

$$(\boldsymbol{\phi}, \boldsymbol{\theta}) = \max_{\phi,\theta} \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] \tag{A.6}$$

In order to learn disentangled representations of the generating factors, we need a constraint that encourages the distribution of the code to approach a prior $p_\theta(\mathbf{z})$, embodying the notion of disentanglement and independence. This leads to the constrained optimisation problem

$$(\boldsymbol{\phi}, \boldsymbol{\theta}) = \max_{\phi,\theta} \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] \text{ subject to } D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z})) < \epsilon \tag{A.7}$$

where ϵ specifies the strength of the imposed constraint. Such kind of problems are discussed in the CST, Part II, Machine Learning and Bayesian Inference course and can be solved by introducing a Lagrange multiplier $\beta \geq 0$ and maximising the Lagrangian

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - \beta\, D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z})) \tag{A.8}$$

As before, we fix $p_\theta(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$, as it implies no correlation between the elements of $\mathbf{z}$.

Appendix B

Implemented Autoencoder Architectures

Dataset	Architecture	Layers
ShapesSet	Fully Connected	encoder: 4096 - 1200 - 1200 - 10 decoder: 10 - 1200 - 1200 - 1200 - 4096 code dimension: 10
ShapesSet	Convolutional	encoder: 64x64x1 - 32x32x16 - 16x16x32 - 1024 - 10 decoder: 10 - 1024 - 8x8x64 - 16x16x16 - 32x32x4 - 64x64x1 code dimension: 10
MNIST	Fully Connected	encoder: 784 - 512 - 512 - 16 decoder: 16 - 512 - 512 - 512 - 784 code dimension: 16
MNIST	Convolutional	encoder: 28x28x1 - 14x14x8 - 7x7x16 - 512 - 16 decoder: 16 - 512 - 7x7x16 - 14x14x8 - 28x28x4 - 28x28x1 code dimension: 16

Table B.1: Autoencoder architectures that were constructed and evaluated during the project. Activation functions: ReLU for the encoder layers, tanh for the decoder layers. The code is considered to be the last layer of the encoder and the first layer of the decoder.

Appendix C

Dataset Generation for the Classifier Measuring Disentanglement

Algorithm 7 Generating a balanced dataset used for the training and the evaluation of the classifier, designed to measure autoencoders' disentanglement.

Require: N – required number of datapoints
Require: *autoencoder* – trained autoencoder model to be evaluated

procedure GenerateClassifierDataset(N, *autoencoder*, *shape*)
  for $x \leftarrow 0$ **to** N **do**
    $\boldsymbol{f}_1^{shape} = shape$
    $\boldsymbol{f}_1 \leftarrow$ randomly sampled values for the rest of the generating factors
    $changeFactor \leftarrow$ randomly chosen factor which will be altered
    $f' \leftarrow$ a new, different randomly sampled value for $changeFactor$
    $\boldsymbol{f}_2 \leftarrow \left(\boldsymbol{f}_1 \setminus \boldsymbol{f}_1^{changeFactor}\right) \cup f'$
    $img_1 =$ GenerateImage($\boldsymbol{f}_1$)
    $img_2 =$ GenerateImage($\boldsymbol{f}_2$)
    $\mathbf{z}_1^{\mu} = autoencoder.encoder(img_1)$
    $\mathbf{z}_2^{\mu} = autoencoder.encoder(img_2)$
    emit datapoint $\left(\frac{|\mathbf{z}_1^{\mu} - \mathbf{z}_2^{\mu}|}{\max(|\mathbf{z}_1^{\mu} - \mathbf{z}_2^{\mu}|)}, changeFactor\right)$
  end for
end procedure

procedure GenerateBalancedClassifierDataset(N, *autoencoder*)
  for $shape \leftarrow \{\square, \bigcirc, \triangle\}$ **do** ▷ Generate balanced dataset again
    GenerateClassifierDataset($N/3$, *autoencoder*, *shape*)
  end for
end procedure

Appendix D

Low-capacity linear classifier

Classification is a supervised learning problem, i.e. the dataset is labeled: $\{(\mathbf{x}'_1, y'_1), (\mathbf{x}'_2, y'_2), \ldots, (\mathbf{x}_N, y'_N)\}$. The goal is given a new datapoint $\mathbf{x}$, to determine the most likely class y it belongs to.

The simple classifier I implemented is a fully connected neural network with input and output layers and no hidden layers. The output layer is passed through the $softmax$ activation function (Figure D.1).

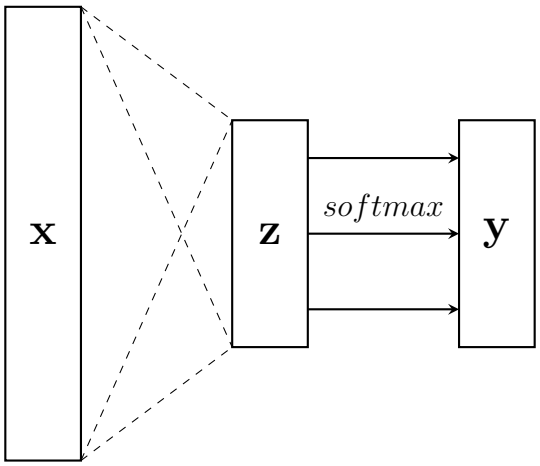


Figure D.1: A simple, low-capacity linear classifier, that was used in the computation of autoencoders' disentanglement.

The $softmax$ activation function is such that

$$\mathbf{y} = softmax(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_k \exp(z_k)} \tag{D.1}$$

That is, $\mathbf{y}$ can be regarded as a probability distribution, indicating how likely the input is to be in each class. An input $\mathbf{x}$ is then classified to the class with the highest probability.

If the original data labels are represented as one-hot vectors $[0, \ldots, 0, 1, 0, \ldots, 0]$ (which is also a valid probability distribution), where only the index of the true class is switched on and everything else is 0, the classifier can be trained with the cross-entropy loss function, minimising

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \mathbf{y}'_i \cdot \log \mathbf{y}_i + (1 - \mathbf{y}'_i) \cdot \log(1 - \mathbf{y}_i) \tag{D.2}$$

The vector operations are element wise, $\mathbf{y}'_i$ is the one-hot representation of the true label y'_i, and $\mathbf{y}_i = softmax(\mathbf{W}\mathbf{x}_i + \mathbf{b})$ is computed by the network.

The classifier was unit tested against known results when trained on the MNIST dataset (Figure D.2).

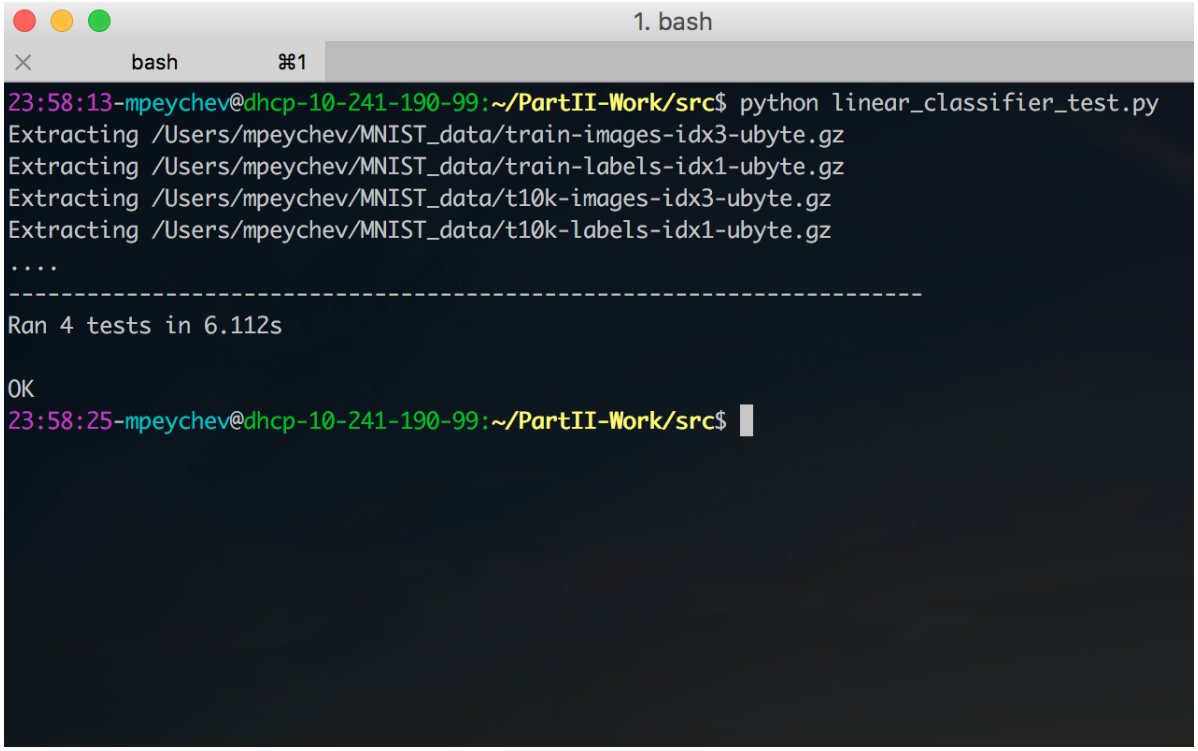


Figure D.2: Executing the unit tests for the linear classifier.

Appendix E

Project Proposal

Momchil Peychev
Girton College
mp739

Project Proposal

Computer Science Tripos, Part II

Experimental Study on the Properties of Disentangled Autoencoders

20 October 2016

Project Originators:
Petar Veličković
Momchil Peychev

Project Supervisors:
Petar Veličković
Dr Pietro Liò

Director of Studies:
Chris Hadley

Project Overseers:
Dr Ian Wassell
Prof Larry Paulson

Introduction and Description of the Work

Despite their recent relative success and popularity, a major issue which is raised about neural networks is their opacity. It is difficult to investigate their exact learning process and, in particular, to reason about both the learnt weights and the underlying high-level features. This is a major hindrance for deploying neural networks in safety-critical systems (e.g. self-driving cars) for the moment. After training a deep neural network, we typically cannot define an easily interpretable decision rule (e.g. for controlling the vehicle in the self-driving car example) out of the learnt parameters and therefore, in case of an accident, it would be very hard to establish whether the problem is in malicious input data, implementation errors, or the proposed model simply not being complex enough to capture the underlying process.

This project will mainly focus on analysing the latent features learnt by the network. I will be particularly interested in the structure of autoencoders [1] (Fig. 1). The classic autoencoder is an artificial neural network composed of two parts - an encoder and a decoder. The encoder receives input features in its input layer and has to produce a code, in the form of a compressed vector representation, out of them. Then the decoder takes the code and aims to reproduce the original features with minimal loss of information. The entire system's parameters are trained simultaneously. Intuitively, autoencoders are suitable for dimensionality reduction and for learning generative models of data.

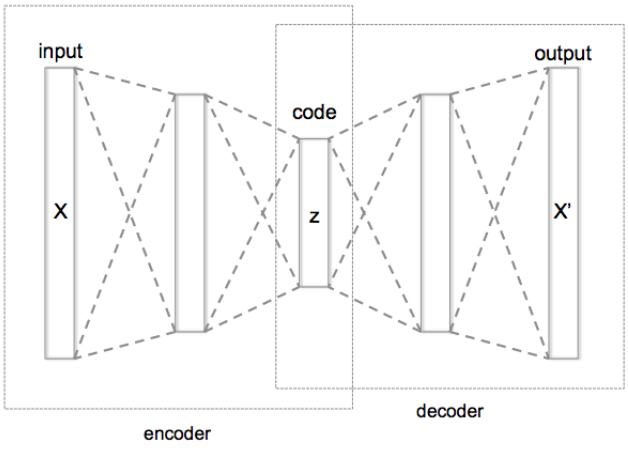


Figure 1: Example autoencoder architecture distinguishing the encoder and the decoder parts. The common layer they share represents the encoded representation of the features.

Recently, the notion of disentanglement has been introduced [2], aiming for interpretable representation and encoding of the input features. A disentangled autoencoder, already proposed in literature, will be considered and implemented with TensorFlow as a widespread and well-established open-source library for numerical computation and the preferred tool for quick and easy neural network mocking. Apart from implementing the software, my goal is to also build a generalised and flexible test-bed environment. In this way, the main

contribution of my core project will be to test the autoencoder's behaviour upon varying its disentanglement parameter. Time permitting, various extensions of the project might be proposed, one of them being to employ disentangled autoencoders in classification problems. To the best of my knowledge, they have not been used for such tasks yet since the disentanglement process comes with the tradeoff of "distracting" the network from being as discriminative as possible. I would like to examine if there exists an equilibrium point where some of the extracted features can be reasoned about and at the same time the classification does not suffer too much from that.

Starting Point

To the best of my knowledge, there is no available open-source implementation that will form a significant basis of my project, so the software development process will begin from scratch. I assume my starting point to be the material covered by the courses in the Computer Science Tripos, with the following being particularly relevant to my future work:

∼ **Artificial Intelligence I** Provides an introduction to the fundamental algorithms in artificial intelligence (AI). The machine learning part of the course presents neural networks and the backpropagation algorithm, which are central to this project.

∼ **Machine Learning and Bayesian Inference** With the project primarily sitting at the Machine Learning area, this course is supposed to serve as a solid background and reference point for any work to be undertaken. It builds on the AI I material covering more advanced Machine Learning ideas. The Deep networks lecture should be tightly connected to the purpose of the project whereas the discussion on classification algorithms can also prove to be potentially useful.

Although not directly applicable, **Mathematical Methods (I/II/III)**, **Mathematical Methods for Computer Science** and **Information Theory** introduce the required mathematical tools to enable me make progress with the models described in the literature which are going to be considered over the project work.

At the time of writing the proposal, Information Theory is an ongoing course and Machine Learning and Bayesian Inference is scheduled for Lent term, which means I will have to familiarise myself with the relevant parts of the course in advance. Inspiration for the project was drawn from [2] but its detailed study will start after the current proposal is firmly accepted (as indicated in the timetable). The same is true for any other publication or open-source software I could utilise.

Substance and Structure of the Project

After the informal general description of autoencoders presented in the Introduction section, what follows here is a more formal mathematical definition of their model.

Autoencoders can be regarded as neural networks with an input layer, an output layer and one or more hidden layers. Let's denote the input feature vector given to the input layer with $\mathbf{x}$. Unlike the classical case where neural networks are trained to predict a certain target value $\mathbf{y}$ given $\mathbf{x}$, autoencoders' goal is to reconstruct their own input $\mathbf{x}'$ as accurately as possible. Therefore, the input and output layers of autoencoders are of the same size. As mentioned before, they are composed by combining two components:

- encoder $\phi : \mathbb{R}^{|\mathbf{x}|} \to \mathbb{R}^n$, and
- decoder $\psi : \mathbb{R}^n \to \mathbb{R}^{|\mathbf{x}|}$.

The encoder and the decoder can be considered partially independent since the encoder produces compressed vector representation $\mathbf{z} = \phi(\mathbf{x})$ out of the input features, whereas the decoder takes the code $\mathbf{z}$ as an input and returns $\mathbf{x}' = \psi(\mathbf{z})$. It can be seen that $|\mathbf{z}| = n$, hence n is the dimension of the desired code. $\mathbf{z}$ is the output on an intermediate hidden layer of the network and is the only common part between the two sections when the network is trained. Classically, the transitions ϕ and ψ are chosen to minimise the sum of the squared errors defined by $\mathcal{L}(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|^2$. However, different variations of the model exist. For instance, the denoising autoencoder receives a partially corrupted input and is trained to reconstruct the original undistorted features.

The disentangled autoencoder presented in [2] aims to obtain such codes in which the statistically independent features are encoded separately. In this way, certain numbers in the code $\mathbf{z}$ will carry valuable meaning about properties of the input features and small perturbations of the code shall result in predictable changes in the decoder output. This is achieved in [2] by modifying the cost function of the autoencoder, adding a scaled disentanglement term $-\beta D_{KL}(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}))$ where D_{KL} stands for the standard Kullback-Leibler divergence and q and p are probability distributions. The authors of [2] also present a way to quantify the disentanglement.

The primary goal of this project is to reproduce the disentangled autoencoder structure and to measure the level of disentanglement with respect to changes of the coefficient β in the cost function.

To do so, the work is split in five sections:

- **Preparation** – background reading and familiarisation with the TensorFlow open-source library. At the end of this point I should have successfully trained and tested several simple neural network models, and therefore become ready for implementing the autoencoder architecture.
- **Core project implementation**

  - Implementing a flexible autoencoder architecture, following the description provided in scientific literature.
  - Creating an extensive test-bed environment to support the forthcoming evaluation.

  This stage will be considered successful upon reproducing known results of similar kinds from documented studies. In case of any significant discrepancies, these should be well-explained and the probable reasons for them must be established. This process will be backed by a unit testing suite.

- **Core project evaluation** – evaluation of the project implementation described above. The first step towards this will be to perform the experiments described in literature and to replicate known results about selected values of β. A natural direct corollary will be to study the effects of intermediate values of the coefficient. This will be done by running the constructed autoencoder varying the coefficient β in the cost function and tracking how the disentanglement level changes.

- **Dissertation** – all the work on the project and the executed studies should be documented in a clear, concise and logical way.

- **Extensions** – time permitting, the disentangled features may be evaluated on classification tasks to see if there is relation between the disentanglement and the discriminative ability of autoencoders. Basic and denoising autoencoders might be considered. This extension will only be implemented and evaluated after successful completion of the core project parts.

Success Criteria

The project will be considered successful upon completion of the **Core project implementation**, **Core project evaluation** and **Dissertation** sections. Precisely:

- Disentangled autoencoder must be implemented and tested against test cases similar to those provided in relevant literature or academic papers.
- A flexible test-bed should be implemented in support of the project evaluation.
- The effects of the autoencoder cost function coefficient on the disentanglement level must be evaluated.
- Finally, a dissertation should be produced describing the work undertaken during the project.

The project **Extensions** should also be achievable, but they are not necessary in order to treat the project as successful.

Resources Required

I intend to use my own machine (2.7 GHz Intel Core i5, 8 GB RAM, running macOS Sierra 10.12) for developing the project work. I accept full responsibility for this machine and I will make the following contingency plans to protect myself against hardware and/or software failure:

- Revision control of the project's codebase and dissertation using `git` and committing the files to remote repositories in GitLab;
- Synchronising the project folders with my Google Drive and Dropbox file space;
- Frequent backups of the project files and folders on an external 2 TB HDD.

This way, in case of technical issues with my laptop, the work can continue on a MCS machine or a virtual instance.

The only open-source library I intend to use thus far is TensorFlow. This would require the software development to be done in Python, which means other libraries might also be easily utilised, in the event that this is found to be useful.

The software might eventually be deployed on GPU machines provided by the Computational Biology Group in the Computer Laboratory.

An important component of the project will be the training and evaluation of the developed models, which means that data will be required. However, since the core goal is to replicate the experiments described in literature, I am in no way locked to any particular dataset. Therefore, if no suitable data is found, artificial datasets with white shapes on black background as well as Atari stills (similar to those used in [2]), can easily be generated programmatically. There exist popular image datasets designed specifically as machine learning benchmarks (e.g. MNIST - `http://yann.lecun.com/exdb/mnist` or CIFAR-10 - `https://www.cs.toronto.edu/~kriz/cifar.html`), and the disentangled classifier developed as a potential project extension can be compared to a classical neural network against them.

References

1. Baldi P., Autoencoders, Unsupervised Learning, and Deep Architectures
2. Higgins I., Matthey L., Glorot X., Pal A., Uria B., Blundell C., Mohamed S., Lerchner A., Early Visual Concept Learning with Unsupervised Deep Learning

Timetable and Milestones

The project work is scheduled in 14 two-week sprint periods. The high-level goals are to complete the core project implementation and evaluation until 1 January 2017 and

the extensions until the end of Lent Term. The dissertation should be produced over the spring holiday and no project work is planned during Easter Term, which should give me enough time to revise and prepare for exams. However, in case of any extreme or unpredictable situations, I am sparing a three-week buffer before the submission deadline to fix any potential problems that may appear.

Sprint 0: *5 October – 9 October*

→ Meetings with Petar Veličković and Dr Pietro Liò.

→ Deciding on a project topic; writing Phase 1 Report.

Milestone: Phase 1 Report ready for submission.

Sprint 1: *10 October – 23 October*

→ Working on the project proposal.

→ Setting up the contingency schemes as described above.

Deadlines:
Phase 1 Report (10 October), Draft Proposal (14 October)
Project proposal submission (21 October)

Milestone: Project proposal is submitted.

Sprint 2: *24 October – 6 November*

→ Reading academic papers and understanding the details behind the autoencoder structure and architecture.

→ Getting familiar with TensorFlow and training basic models.

Milestone: A simple neural network should be successfully deployed and tested against classical problems described in literature.

Sprint 3: *7 November – 20 November*

→ Further exploration of the mathematical model of autoencoders.

→ Basic autoencoder implementation in TensorFlow.

Milestone: Must have a running autoencoder working correctly, waiting to be evaluated.

Sprint 4: *21 November – 4 December*

→ Building a flexible and comprehensive test-bed to be used for evaluation.

Milestone: None.

Sprint 5: *5 December – 18 December*

→ Buffer period for bug fixing and finishing any pending milestones so far.

→ Should be able to proceed with project evaluation.

Milestone: Basic autoencoder functionality running normally in a way suitable to be evaluated. (**Core project implementation completed**).

Sprint 6: *19 December – 1 January*

→ Running tests and experiments on datasets mentioned in the Resources section.

→ Evaluating the effects of the autoencoder cost function parameters to its disentanglement level.

Milestone: (**Core project evaluation completed**).

Sprint 7: *2 January – 15 January*

→ Writing progress report. Sending it to supervisors for comments and corrections.

Milestone: Completing the work on the progress report.

Sprint 8: *16 January – 29 January*

→ Finishing the progress report.

→ Preparing the slides for the presentation.

Milestone: Progress report is sent to overseers, presentation slides are done.

Sprint 9: *30 January – 12 February*

→ Presentation rehearsal.

→ Planning and setting up the ground for the extensions discussed above.

Deadlines:
Progress Report (3 February)
Progress Report Presentations (9/10 - 13/14 February)

Milestone: Ready to give progress report presentation.

Sprint 10: *13 February – 26 February*

→ Extending the autoencoder structure to be able to perform classification tasks.

Milestone: (**Extension implementation completed**).

Sprint 11: *27 February – 12 March*

→ Evaluating the extension. Studying the relation between the disentanglement and the discriminative abilities of autoencoders.

Milestone: (**Extension evaluation completed**).

Sprint 12: *13 March – 26 March*

→ Start writing the dissertation.

Milestone: Completion of the Introduction and Preparation chapters of the dissertation.

Sprint 13: *27 March – 9 April*

→ Continue working on the dissertation.

Milestone: Implementation, Evaluation and Conclusion chapters completed. Dissertation draft is sent to supervisors for comments and corrections.

Sprint 14: *10 April – 23 April*

→ Dissertation editting and refactoring.

Milestone: Work on dissertation completed, ready to be submitted (**Dissertation completed**).

Sprint 15: *24 April – 19 May*

→ A buffer period in case something in process scheduled so far goes wrong.

Deadline: Dissertation submission (19 May)