Master's thesis

November 22, 2023

On variational autoencoders: theory and applications

Maksym Sevkovych

Registration number: 3330007

In collaboration with: Duckeneers GmbH

Inspector: Univ. Prof. Dr. Ingo Steinwart

Recently in the realm of machine learning, the power of generative models has revolutionized the way we perceive data representation and creation. This thesis focuses on the captivating domain of Variational Autoencoders (VAEs), a cutting-edge class of machine learning models that seamlessly combine unsupervised learning and data generation. In the course of this thesis we embark on an expedition through the intricate architecture and mathematical elegance that underlie VAEs.

By dissecting the architecture of VAEs, we show their role as both proficient data compressors and imaginative creators. As we navigate the landscapes of latent spaces and probabilistic encodings, we uncover the essential mechanisms driving their flexibility. Applications of VAEs extend from anomaly detection to image generation. However, we will focus on the latter.

Contents

1 Preliminary **3**
1.1 Neural networks 3
1.2 Training of neural networks 5
1.3 Neural networks in computer vision 8

2 Autoencoders **14**
2.1 Conceptional ideas 14
2.2 Training of autoencoders 18
2.3 Applications 19

3 Variational Autoencoders **21**
3.1 What are variational autoencoders? 21

References **23**

1 Preliminary

In order to fathom the topic of variational autoencoders or even autoencoders in general, we need to consider a couple of preliminary ideas. Those ideas consist mainly of neural networks and their optimization - usually being called training. In this chapter, we will tackle the conceptional idea of how to formulate neural networks in a mathematical way and furthermore, we will consider a couple of useful operations that neural networks are capable of doing. Then, we will take a look at some strategies of training neural networks. Lastly, we will consider neural networks operating on images. This discipline of machine learning is usually referred to as computer vision.

1.1 Neural networks

The idea of artificial neural networks originated from analysing mammal's brains. An accumulation of nodes - so called neurons, connected in a very special way that fire an electric impulse to adjacent neurons upon being triggered and transmit information that way. Scientists tried to mimic this natural architecture and replicate this mammal intelligence artificially. This research has been going for almost 80 years and became immensely popular recently through artificial intelligences like OpenAI's ChatGPT or Google's Bard for the broad public. But what actually is a neural network? What happens in a neural network? Those are very interesting and important questions that we will find answers for.
As already mentioned, neural networks consist of single neurons that transmit information upon being „triggered". Obviously, triggering an artificial neuron can't happen the same way as neurological neurons are being triggered through stimulus. Hence, we need to model the triggering of a neuron in some way. The idea is to filter information that does not exceed a certain stimulus threshold. This filter is usually being called activation function. Indeed, there are lots of ways of modelling such activation functions and it primarily depends on the specific use-case what exactly the activation function has to fulfil. Therefore, we define activation functions in the most general way possible.
TODO: Give a formal reference to neural networks somewhere?

Definition 1.1.1. A non-constant function $\varphi : \mathbb{R} \to \mathbb{R}$ is called an **activation function** if it is continuous.

Even though there are lots of different activation functions, we want to consider mainly the following ones, see [1, chapter 6].

Example 1.1.2. The following functions are activation functions.

Rectified Linear Unit (ReLU): $\varphi(t) = \max\{0, t\}$,

Leaky Rectified Linear Unit (Leaky ReLU): $\varphi(t) = \begin{cases} \alpha t, & t \leq 0, \\ t, & t > 0. \end{cases}$

Sigmoid: $$\varphi(t) = \frac{1}{1+e^{-t}}.$$

Now, having introduced activation functions we can introduce neurons.

Definition 1.1.3. Let $\varphi : \mathbb{R} \to \mathbb{R}$ be an activation function and $w \in \mathbb{R}^k$, $b \in \mathbb{R}$. Then a function $h : \mathbb{R}^k \to \mathbb{R}$ is called **φ-neuron** with weight w and bias b, if

$$h(x) = \varphi\left(\langle w, x\rangle + b\right), \qquad x \in \mathbb{R}^k. \tag{1.1}$$

We call $\theta \coloneqq (w, b)$ the parameters of the neuron h.

If we arrange multiple neurons next to each other, we can define a layer consisting of neurons. This way we can expand the architecture from one to multiple neurons.

Definition 1.1.4. Let $\varphi : \mathbb{R} \to \mathbb{R}$ be an activation function and $W \in \mathbb{R}^{m\times k}$, $b \in \mathbb{R}^m$. Then a function $H : \mathbb{R}^k \to \mathbb{R}^m$ is called **φ-layer** of width m with **weights** W and **biases** b if for all $i = 1, \dots, m$ the component function h_i of H is a φ-neuron with weight $w_i = W^\top e_i$ and bias $b_i = \langle b, e_i\rangle$, where e_i denotes the standard ONB of $\mathbb{R}^m$.
If we consider $\widehat{\varphi} : \mathbb{R}^k \to \mathbb{R}$ as the component-wise mapping of $\varphi : \mathbb{R} \to \mathbb{R}$, meaning $\widehat{\varphi}(v) = (\varphi(v_1), \dots, \varphi(v_k))$, we can write the φ-layer $H : \mathbb{R}^k \to \mathbb{R}^m$ as

$$H(x) = \widehat{\varphi}(Wx + b), \qquad x \in \mathbb{R}^k. \tag{1.2}$$

In the following, we will denote the weights W and biases b as **parameters of the neural network** $\theta \coloneqq (W, b)$.

Finally, we can introduce neural networks as an arrangement of multiple neural layers.

Definition 1.1.5. Let $L \in \mathbb{N}$, $\varphi_1, \dots, \varphi_L$ be activation functions and $H_1, \dots, H_L$ be φ_i-layers with parameters $\theta_i = (W_i, b_i)$ for all $i \in \{1, \dots, L\}$. Furthermore, let $\theta = (\theta_1, \dots \theta_L)$, $\varphi = (\varphi_1, \dots, \varphi_L)$ and $d_1, \dots, d_{L+1} \in \mathbb{N}$.
Then we define a **φ-deep neural network** of depth L with parameters $\theta \in \Theta$ as

$$\begin{aligned} f_{\varphi,L,\theta} : \mathbb{R}^{d_1} &\to \mathbb{R}^{d_{L+1}} \\ x &\to H_L \circ \dots \circ H_1(x), \qquad x \in \mathbb{R}^{d_1}, \end{aligned} \tag{1.3}$$

where each $H_i : \mathbb{R}^{d_i} \to \mathbb{R}^{d_{i+1}}$ is a φ_i-layer. Ultimately, d_1 describes the input dimension and d_{L+1} the output dimension of the neural network.
Lastly, we will write $f \coloneqq f_{\varphi,L,\theta}$, if the activation function φ, the depth L and the parameters θ are clear out of context.

One may realize, that the neural network is defined in a way that the activation function may vary in each layer. However, in most applications the input layer and the hidden layers ($i \in \{1, \dots, L-1\}$) share the same activation function. The last layer, often referred to as output layer, usually has a different activation function. This activation function may as well be the identity function.
A visual representation of a neural network can be found in figure 1.1

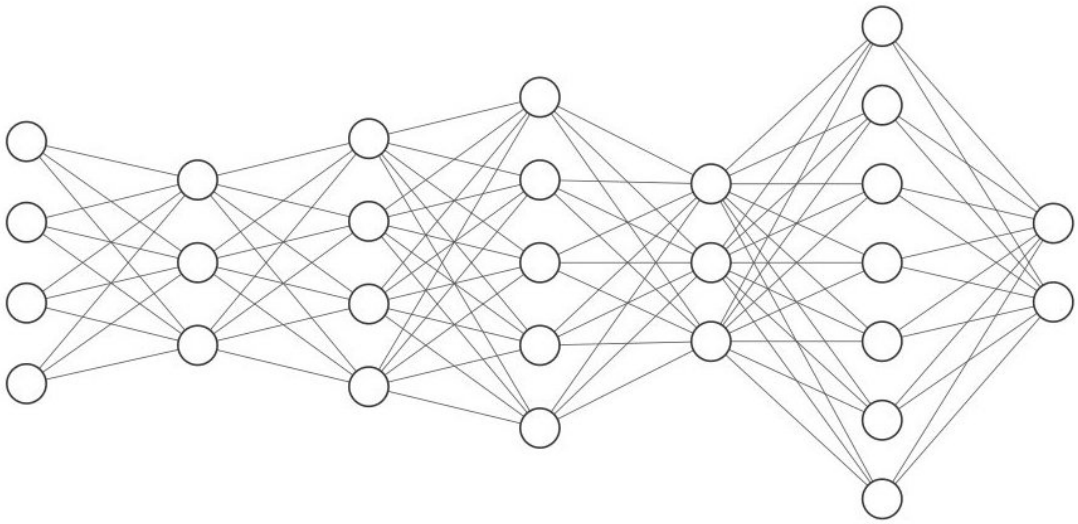

Figure 1.1: A neural network with input $x \in \mathbb{R}^4$ and output $y \in \mathbb{R}^2$. The five hidden layers have dimensions 3, 4, 5, 3 and 7 respectively. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

1.2 Training of neural networks

Since we now know what neural networks are, we want to discuss how to tune them to a specific problem. This procedure is usually referred to as training of a neural network. There are many approaches of how to train a neural network. However, all of them require the definitions of quantities called loss function and risk function. The loss function is a function that measures the point-wise error of a neural network, or any other prediction function in general. This is fundamental in supervised learning. In contrast, the risk function is a function that measures the error with regard to a probability measure or an observed data set.

Definition 1.2.1. Let $d, n \in \mathbb{N}$ and $X \subseteq \mathbb{R}^d$ and $Y \subseteq \mathbb{R}^n$, that we will refer to as input and output space and $t \in \mathbb{R}^n$ be a prediction for $y \in Y$.
Then we define a **supervised loss function** $L : X \times Y \times \mathbb{R}^n \to [0, \infty)$ as a measurable function that compares a true value $y \in Y \subset \mathbb{R}^n$ to a predicted value $\widehat{y} = t$ in a suitable way.

The recently introduced loss function allows us to compare a true label to a prediction. However, since we only require it to be measurable, the function is defined very general. This is solely, because one may be interested in different loss functions in different settings, since the choice of a specific loss functions is a crucial aspect of learning theory, as it directly impacts the behaviour of learning algorithms.
We want to consider a couple of important examples for loss functions, for further reading please take a look at [1, chapter 4.3].

Example 1.2.2. Let $y \in Y \coloneqq \mathbb{R}$ and $t \in \mathbb{R}$, then the following functions are loss functions.

Squared Error Loss: $L(x, y, t) = |y - t|^2$,

Linear Error Loss: $L(x, y, t) = |y - t|$.

Now, let $y \in Y \coloneqq \mathbb{R}^n$ and $t \in \mathbb{R}^n$ with $n > 1$. Then, we consider the squared error loss and the linear error loss pointwise:

Squared Error Loss: $L(x, y, t) = \sum_{i=1}^{n} |y_i - t_i|^2$,

Linear Error Loss: $L(x, y, t) = \sum_{i=1}^{n} |y_i - t_i|$,

where $y = (y_1, \ldots, y_n)$ and $t = (t_1, \ldots, t_n)$.

However, loss functions only quantify the disparity between one predicted outcome and one actual value. In order to compare the predictions over a wider range of data points, we need to introduce another quantity, the risk function.

Definition 1.2.3. Let X, Y be input and output spaces, L be a loss function and P be a probability measure on $X \times Y$.
Then we define the **risk of the function** f with regard to a loss function L as

$$\mathcal{R}_{L,\mathrm{P}}(f) = \int_{X \times Y} L\left(x, y, f(x)\right) d\,\mathrm{P}(x, y).$$

In the following, we will denote this as the **L-risk of** f.

Considering applications, one usually wants to compute the risk with regard to observed data instead of a probability measure, since it usually is unknown. In this case, the general risk function becomes more tangible, as we see in the following definition.

Definition 1.2.4. Let X, Y be arbitrary input and output spaces, $D = ((x_1, y_1), \ldots, (x_k, y_k))$ be a dataset consisting of $k \in \mathbb{N}$ data points. Furthermore, let L be a loss function and f be an arbitrary prediction function.
Then we define the **empirical risk function** as

$$\mathcal{R}_{L,D}(f) = \frac{1}{k} \sum_{i=1}^{k} L\left(x_i, y_i, f(x_i)\right). \tag{1.4}$$

We will write $\mathcal{R} \coloneqq \mathcal{R}_{L,D}$ unless unclear in the given context.

With the previous definitions, we can return to considering the training of neural networks. There are many possible techniques and strategies. However, most of them rely on iteratively finding the gradient - the direction of greatest ascent of the risk function. In the following we want to consider a couple of popular algorithms that are used to train neural networks.

Theorem 1.2.5. *Let $(\gamma_t)_{t \in \mathbb{N}}$ be a sequence with $\gamma_t \to 0$. Furthermore, let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuous, convex and differentiable function. Furthermore, let $x^{(t)}$ denote the t-th iterate of the **gradient descent algorithm** defined by*

$$x^{(t+1)} = x^{(t)} - \gamma_t\, \partial_x f(x^{(t)}), \tag{1.5}$$

with a suitable initial guess $x^{(0)} \in \mathbb{R}^n$.
Then the algorithm converges to the global minimum $f(x^*) \in \mathbb{R}$, meaning

$$x^* \coloneqq \arg\min_{x \in \mathbb{R}^n} f(x) = \lim_{t \to \infty} x^{(t)}.$$

Lastly, if the function f is strictly convex, then the global minimum $f(x^*) \in \mathbb{R}$ is unique.

Proof. If the step size is sufficiently small such that the iterate is contained in the sphere around x^* with radius $d\left(x^*, x^{(t)}\right)$, the iterate $x^{(t+1)}$ is bound by a sphere around x^* with radius $d\left(x^*, x^{(t)} - \gamma_t \nabla_x f(x^{(t)})\right) < d\left(x^*, x^{(t)}\right)$, since

$$\begin{aligned} d\left(x^*, x^{(t)}\right) &\geq d\left(x^*, x^{(t+1)}\right) \\ &= d\left(x^*, x^{(t)} - \gamma_t \nabla_x f(x^{(t)})\right). \end{aligned}$$

Hence, the distance $d(x^*, x^{(t)})$ becomes smaller in each iteration, due to the convexity of f, with

$$\lim_{t\to\infty} d\left(x^*, x^{(t)}\right) = 0,$$

since we know that $\mathbb{R}^n$ is a Banach space and $\left(d\left(x^*, x^{(t)}\right)\right)_{t\in\mathbb{N}}$ is a converging sequence by construction.
It is left to show, that if the function f is strictly convex, then the global minimum $f(x^*) \in \mathbb{R}$ is unique. This assertion holds, since if there were two global minima $f\left(x_1^*\right), f\left(x_2^*\right)$ with $x_1^* \neq x_2^*$. Now consider $x' \coloneqq \frac{x_1^* + x_2^*}{2}$, a point between x_1^* and x_1^*. Since f is assumed to be strictly convex, this leads to

$$f\left(x'\right) = f\left(\frac{1}{2}x_1^* + \frac{1}{2}x_2^*\right) < \frac{1}{2}f\left(x_1^*\right) + \frac{1}{2}f\left(x_2^*\right) = f\left(x_1^*\right) = f\left(x_2^*\right).$$

This would contradict the assumption that $f\left(x_1^*\right), f\left(x_2^*\right)$ are minima, especially global minima. Hence, the assertion holds. □

In order to apply theorem 1.2.5 to neural networks, we have to consider how to actually compute the gradient of the empirical risk function, where we consider a neural network as prediction function.

Lemma 1.2.6. *Let $f_\theta : \mathbb{R}^d \to \mathbb{R}$ be a neural network with parameters $\theta \in \Theta$, arbitrary depth $L_n \in \mathbb{N}$ and arbitrary activation function φ. Furthermore, let D be a dataset of length $k \in \mathbb{N}$ and L be an arbitrary loss function.*
The gradient of the risk function $\mathcal{R}(\cdot)$ with regard to the neural network f_θ and thus the parameters θ look as follows

$$\partial_\theta \mathcal{R}\left(f_\theta\right) = \frac{1}{k}\sum_{i=1}^{k} \partial_\theta L\left(x_i, y_i, f_\theta(x_i)\right).$$

Hence, it is the average of gradients in all data points $(x_i, y_i) \in D$.

Proof. To prove the assertion we simply use the definition 1.2.4 of the empirical risk function and consider the linearity property of derivatives.

$$\begin{aligned} \partial_\theta \mathcal{R}\left(f_\theta\right) &= \partial_\theta \frac{1}{k}\sum_{i=1}^{k} L\left(x_i, y_i, f_\theta(x_i)\right) \\ &= \frac{1}{k}\sum_{i=1}^{k} \partial_\theta L\left(x_i, y_i, f_\theta(x_i)\right). \end{aligned}$$

□

With the previous definitions and results we can formulate the gradient descent algorithm for a neural network.

Corollary 1.2.7. *Let $f_\theta : \mathbb{R}^d \to \mathbb{R}$ be a neural network with parameters $\theta \in \Theta$, arbitrary depth $L \in \mathbb{N}$ and arbitrary activation function φ. Let $(\gamma_t)_{t\in\mathbb{N}}$ be a sequence with $\gamma_t \to 0$ and D be a dataset of length $k \in \mathbb{N}$.*
Then one can train the neural network f_θ with the gradient descent algorithm proposed in theorem 1.2.5. In this setting, the algorithm looks as follows

$$\theta^{(t)} = \theta^{(t-1)} - \gamma_{t-1}\, \partial_\theta \mathcal{R}\left(f_{\theta^{(t-1)}}\right),$$

where the gradient can be computed as in lemma 1.2.6

TODO: Name some properties (convergence, rate, etc.) and reference them

This is a valuable result, since this way one can iteratively optimize any convex function. Such iterative methods are powerful in numerical settings, where one could use a machine to compute the result. However, there is one problem: in many practical cases it is way to expensive to compute the gradient with regard to the whole dataset, if the dataset becomes significantly large. This lead to a bunch of approaches on how to make this algorithm more efficient, one of those being the stochastic gradient descent algorithm.

Theorem 1.2.8. *Let $f_\theta : \mathbb{R}^d \to \mathbb{R}$ be a neural network with parameters $\theta \in \Theta$, arbitrary depth $L \in \mathbb{N}$ and arbitrary activation function φ. Let $(\gamma_t)_{t\in\mathbb{N}}$ be a sequence with $\gamma_t \to 0$ and D be a dataset of length $k \in \mathbb{N}$.*
*Then we define the t-th iterate of the **stochastic gradient descent algorithm** by*

$$\theta^{(t)} = \theta^{(t-1)} - \gamma_{t-1}\, \partial_{\theta,i} \mathcal{R}\left(f_{\theta^{(t-1)}}\right), \tag{1.6}$$

with $i \in \{1, \dots, k\}$ and $\partial_{\theta,i} \mathcal{R}\left(f_{\theta^{(t)}}\right)$ denoting the gradient with regard to the i-th data tuple $(x_i, y_i) \in D$.

TODO: Name some properties (convergence, rate, etc.) and reference them

Lastly, we want to consider another powerful optimization algorithm that adapts learning rates based on past gradient magnitudes and momenta - it is called „Adaptive Moment Estimation (ADAM)“, see [2]. See algorithm 1 for pseudo-code of the proposed algorithm.

Algorithm 1 ADAM optimizer

Let $g_t := \nabla_\theta f_t(\theta)$ denote the and $g_t^2 := g_t \odot g_t$ denote the elemen-wise multiplication.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ ▷ Get gradients w.r.t. stochastic objective at timestep t
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ ▷ Update biased first moment estimate
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ ▷ Update biased second moment estimate
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Compute bias-corrected first moment estimate
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Compute bias-corrected second moment estimate
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$
end while
return θ_t ▷ Return Resulting parameters

TODO: Since ADAM optimizers perform best at the current state of the art, it would be nice to see how it works. However, it would need some pages to introduce.. Is it worth it? Probably it is.

1.3 Neural networks in computer vision

Lastly in this chapter, we want to apply the theory of neural networks to the setting we actually are interested in. This setting is usually called computer vision - basically, machine learning that is applied to images and videos. Since we want to apply neural networks to a problem that deals with images, we need to know how to view images from a mathematical perspective. In order to do so, we need to introduce some quantities first. The first important quantity is a pixel. Basically, this is a single point in the image.

Definition 1.3.1. Let $d \in \mathbb{N}$ and $\Psi = \{0, \dots, 255\}$. Then we define a **pixel with d channels** as

$$\psi = (\psi_1, \psi_2, \dots, \psi_d),$$

where for all $i = 1, \dots, d$ holds $\psi_i \in \Psi$. Hence, for each pixel holds $\psi \in \Psi^d$

Remark 1.3.2. We want to distinguish mainly two kinds of pixels. If $d = 1$, we speak of a **black and white pixel**.
If $d = 3$, we speak of an **RGB pixel**. Here, RGB stands for the red, green and blue color channels.

As one may already know, images consist of multiple pixels that are aligned in a grid. We will refer to this grid as a pixel domain.

Definition 1.3.3. Let $M \in \mathbb{N}$ be the **horizontal amount of pixels** and $N \in \mathbb{N}$ be the **vertical amount of pixels**. Then we call the grid Ω defined by

$$\Omega \coloneqq \{1, \dots, M\} \times \{1, \dots, N\} \subset \mathbb{N}^2,$$

the **pixel domain** with the tuple (M, N) being called the **resolution**.

If we now combine the definitions of a pixel and a pixel domain, we can define a mathematical representation of an image - a so called digital image.

Definition 1.3.4. Let $d \in \mathbb{N}$ be the number of channels and Ω a pixel domain with the resolution (M, N). Then we define a **digital image** by

$$\psi = \left(\psi_{ij}\right)_{i,j} = \left(\psi_{ij,1}, \dots, \psi_{ij,d}\right), \qquad (i,j) \in \Omega,$$

where each ψ_{ij} is a pixel with d channels.
Since for each pixel ψ_{ij} holds $\psi_{ij} \in \Psi^d$, we define the **image domain** as $\Psi^{d \times M \times N}$. We will write $\psi \in \Psi_{d,\Omega} \coloneqq \Psi^{d \times M \times N}$ from now on. If the context is clear, we will reduce the notation up to Ψ.

However, since we usually consider floats and not integers in numerical mathematics, we need to consider a way to represent pixels as floats. In order to do that, we introduce the normed image domain.

Definition 1.3.5. Let Ω be a pixel domain and $d \in \mathbb{N}$ a number of channels. Then we call the Ψ', defined as

$$\Psi' \coloneqq \frac{1}{255}\Psi = \left\{\frac{0}{255}, \frac{1}{255}, \dots, \frac{255}{255}\right\},$$

the **normed image domain**. Obviously, it holds $\Psi' \subset [0,1]$.

We realize that we can easily transform images from an ordinary image domain to a normed image domain by dividing each pixel value by 255. Equally, we can transform images the other way around by multiplying each pixel value by 255.
Since neural networks rarely produce a value that is a fraction with denominator 255, we need to find a way to process such values. This we will do in the following lemma.

Lemma 1.3.6. *Let $d \in \mathbb{N}$ be a number of channels and let $p \in [0,1]^d$. Then we can consider p as a pixel by transforming it through*

$$\psi_i \approx 255 \cdot p_i, \qquad i = 1, \dots, d,$$

where we round each entry p_i to an integer.

Proof. Since $p \in [0,1]^d$, we can denote p as

$$p = (p_1, \dots, p_d),$$

where for all $i = 1, \dots, d$ holds $p_i \in [0,1]$.
Hence, if we multiply each p_i by 255, it follows that

$$255 \cdot p_i \in [0, 255].$$

Therefore, it holds that

$$255 \cdot p \in [0, 255]^d.$$

If we now round each entry p_i to an integer, we yield exactly the representation defined in definition 1.3.1. □

Now having formally defined what an image is, we can consider how a neural network operating on images looks like. In order to do this, it is sufficient to consider a single neural layer, since neural networks consist of multiple layers. But first, we need some technical assertions to understand how we can actually feed images into neural networks, since images are somewhat matrices and neural networks often operate on arrays.

Lemma 1.3.7. *Let Ω be a pixel domain with resolution (M, N). Then the image ψ with $d \in \mathbb{N}$ channels defined on $\Psi_{d,\Omega}$ can be represented as an MN-dimensional array instead of an $M \times N$-matrix.*

Proof. Let ψ be a matrix with entries $\psi_{ij} \in \Psi_d$, what follows from the definition of an image 1.3.4. Hence, ψ is an $M \times N$- matrix.
Define the rows of ψ by $\widehat{\psi}_i \coloneqq (\psi_{i1}, \dots, \psi_{iN})$ for all $i \in \{1, \dots, M\}$. Then the matrix representation of the picture ψ can be written as

$$\begin{pmatrix} \psi_{11} & \cdots & \psi_{1N} \\ \vdots & \ddots & \vdots \\ \psi_{M1} & \cdots & \psi_{MN} \end{pmatrix} = \begin{pmatrix} \widehat{\psi}_1 \\ \vdots \\ \widehat{\psi}_M \end{pmatrix}.$$

If we now transpose each $\widehat{\psi}_i$ and keep the same representation, we transform the $M \times N$ matrix into an MN-dimensional array

$$\begin{pmatrix} \widehat{\psi}_1^\intercal \\ \vdots \\ \widehat{\psi}_M^\intercal \end{pmatrix} \in \Psi^{MN}.$$

□

Lemma 1.3.7 allows us to feed images into a neural network by considering them as one large array. However, it still is unclear how the images are processed throughout the neural network. In order to formulate this, we need to define a function operating on images.

Definition 1.3.8. Let Ω_0 and Ω_1 be pixel domains with resolutions (M_0, N_0) and (M_1, N_1) respectively. Furthermore, let $d \in \mathbb{N}$ be an arbitrary number of channels.
Then we define an **image operator** T as the continuous mapping

$$\begin{aligned} T : \Psi_{d,\Omega_0} &\to \Psi_{d,\Omega_1} \\ \psi_0 &\mapsto \psi_1 \coloneqq T\,\psi_0, \end{aligned}$$

where T does not change the number of channels d. Thus, we shorten Ψ_{d,Ω_0} to Ψ_{Ω_0} in the following.

The definition 1.3.8 is quite general, since we do not demand any specific properties from the image operator T whatsoever. Indeed, there are some image operators that are commonly used in computer vision. We will take a look at those in the course of this section.
These technicalities helps us introduce neural networks operating on pictures.

Corollary 1.3.9. *Let φ be an arbitrary activation function and $\widehat{\varphi}$ the component-wise mapping of φ as in 1.1.4 and Ω_0 be an arbitrary pixel domain with resolution $(M_0, N_0) \in \mathbb{N}^2$. Let ψ be an image with number of channels $d \in \mathbb{N}$.*
Then a neural layer that operates on images looks as follows

$$\begin{aligned} H : \Psi_{\Omega_0} &\to \Psi_{\Omega_1} \\ \psi_0 &\mapsto \widehat{\varphi}\,(T\,\psi_0 + b)\,, \end{aligned}$$

where T is an image operator as in definition 1.3.8, b is a bias and Ω_1 a pixel domain with resolution (M_1, N_1).

With the help of corollary 1.3.9 we realise, that each layer H_i of a neural network in a computer vision setting represents an own image space, denoted by Ψ_{Ω_i}. Meaning, that all elements fed to the corresponding layer are images with resolution (M_i, N_i).
Now we want to consider some useful examples of image operators. Firstly, we will take a look at multidimensional convolutions, hence convolutions on images. For more details please look at [1, chapter 9].

Proposition 1.3.10. *Let Ω_0, Ω_1 be pixel spaces with arbitrary but fixed number of channels and resolutions (M_0, N_0) and (M_1, N_1) respectively. Then the **image convolution operator** $T_{conv} : \Psi_{\Omega_0} \to \Psi_{\Omega_1}$ is defined by*

$$\begin{aligned} T_{conv} : \Psi_{\Omega_0} &\to \Psi_{\Omega_1} \\ \psi_0 &\mapsto \psi_1 \coloneqq T_{conv}\,\psi_0\,. \end{aligned}$$

This operator is defined component-wise by

$$(\psi_1)_{ij} \coloneqq (\psi_0 * k)_{ij} = \sum_{m,n=1}^{s} (\psi_0)_{m+i,n+j}\, k_{mn}, \tag{1.7}$$

where $(i, j) \in \Omega_1 \coloneqq \{1, \dots, M_0 - s + 1\} \times \{1, \dots, N_0 - s + 1\}$. Hence, $M_1 = M_0 - s + 1$ and $N_1 = N_0 - s + 1$.
*Furthermore, k is called an s-**convolution kernel**, an image with resolution (s, s) with $s \in \mathbb{N}$.*

Another very useful example is the pooling operator. We will distinguish between average and min- and max-pooling.

Proposition 1.3.11. *Let Ω_0, Ω_1 be pixel spaces with arbitrary but fixed number of channels and resolutions (M_0, N_0) and (M_1, N_1) respectively. Furthermore, $s \in \mathbb{N}$ is called **stride** and $I \coloneqq \{1, \dots, p_1\} \times \{1, \dots, p_2\} \subset \mathbb{N}^2$ with $p_1, p_2 \in \mathbb{N}$ is called **pooling**.*
*Then the **average-pooling operator** $T_{avg} : \Psi_{\Omega_0} \to \Psi_{\Omega_1}$ with stride s and pooling I is defined by*

$$\begin{aligned} T_{avg} : \Psi_{\Omega_0} &\to \Psi_{\Omega_1} \\ \psi_0 &\mapsto \psi_1 \coloneqq T_{avg}\,\psi_0\,. \end{aligned}$$

This operator is defined component-wise by

$$(\psi_1)_{ij} \coloneqq \frac{1}{p_1 p_2} \sum_{(m,n)\in I} (\psi_0)_{m+i,n+j}\,, \tag{1.8}$$

where $(i,j) \in \Omega_1 \coloneqq \{1,\dots,M_0-p_1+1\} \times \{1,\dots,N_0-p_2+1\}$. *Hence,* $M_1 = M_0-p_1+1$ *and* $N_1 = N_0-p_2+1$.

Proposition 1.3.12. *Let* Ω_0, Ω_1 *be pixel spaces with arbitrary but fixed number of channels and resolutions* (M_0,N_0) *and* (M_1,N_1) *respectively. Furthermore,* $s \in \mathbb{N}$ *is called* ***stride*** *and* $I \coloneqq \{1,\dots,p_1\} \times \{1,\dots,p_2\} \subset \mathbb{N}^2$ *with* $p_1,p_2 \in \mathbb{N}$ *is called* ***pooling****.*
Then the ***min-pooling operator*** $T_{min} : \Psi_{\Omega_0} \to \Psi_{\Omega_1}$ *with stride* s *and pooling* I *is defined by*

$$\begin{aligned} T_{min} : \Psi_{\Omega_0} &\to \Psi_{\Omega_1} \\ \psi_0 &\mapsto \psi_1 \coloneqq T_{min}\,\psi_0\,. \end{aligned}$$

This operator is defined component-wise by

$$(\psi_1)_{ij} \coloneqq \min_{(k,l)\in I} (\psi_0)_{kl}\,, \tag{1.9}$$

where $(i,j) \in \Omega_1 \coloneqq \{1,\dots,M_0-p_1+1\} \times \{1,\dots,N_0-p_2+1\}$. *Hence,* $M_1 = M_0-p_1+1$ *and* $N_1 = N_0-p_2+1$.

Proposition 1.3.13. *Let* Ω_0, Ω_1 *be pixel spaces with arbitrary but fixed number of channels and resolutions* (M_0,N_0) *and* (M_1,N_1) *respectively. Furthermore,* $s \in \mathbb{N}$ *is called* ***stride*** *and* $I \coloneqq \{1,\dots,p_1\} \times \{1,\dots,p_2\} \subset \mathbb{N}^2$ *with* $p_1,p_2 \in \mathbb{N}$ *is called* ***pooling****.*
Then the ***max-pooling operator*** $T_{max} : \Psi_{\Omega_0} \to \Psi_{\Omega_1}$ *with stride* s *and pooling* I *is defined by*

$$\begin{aligned} T_{max} : \Psi_{\Omega_0} &\to \Psi_{\Omega_1} \\ \psi_0 &\mapsto \psi_1 \coloneqq T_{max}\,\psi_0\,. \end{aligned}$$

This operator is defined component-wise by

$$(\psi_1)_{ij} \coloneqq \max_{(k,l)\in I} (\psi_0)_{kl}\,, \tag{1.10}$$

where $(i,j) \in \Omega_1 \coloneqq \{1,\dots,M_0-p_1+1\} \times \{1,\dots,N_0-p_2+1\}$. *Hence,* $M_1 = M_0-p_1+1$ *and* $N_1 = N_0-p_2+1$.

2 Autoencoders

Having introduced the basics of neural networks in chapter 1 we can consider a specific architecture of a neural network, a so called autoencoding neural network, or short: autoencoder. The idea of autoencoders is to take a given input, compress the input to a smaller size (usually called encoding) and afterwards, expand it as accurately as possible to the original size again (usually called decoding). Such an architecture is widely used in different applications. For example on social media platforms, where users send images to one another. Instead of sending the original image, which size might very well be a couple of megabytes, the image is firstly being encoded and sent in the compressed representation. Afterwards, the recipient decodes the image to its original representation. This way one has only to transmit the encoded representation, which usually is smaller by magnitudes. Another very important application of autoencoders is in the machine learning field. Most state of the art machine learning models are using autoencoding structures, since it is way more efficient to first encode the data and then run a model on the encoded data. This is quite straight-forward, considering the same argument as in the previous use case - the encoded data being smaller by magnitudes. This way processing the samples can happen much faster compared to the non-encoded data samples and secondly, it makes storing data (on the drive and in memory) much more efficient.
In this chapter we want to consider how to formulate autoencoding neural networks from a mathematical point of view, take a look at some important results and lastly, analyse the theory in multiple applications using Python.

2.1 Conceptional ideas

As already mentioned, an autoencoding neural network first compresses/encodes the input data to a smaller representation. The size of this smaller representation is usually referred to as bottleneck of the autoencoder. Afterwards, the autoencoding neural network expands/decodes the data to its original size. Hence, we can divide these two steps into separate architectures - the encoding and the decoding part of the neural network, which we will formulate separately.
In figure 2.1 we can take a look at a visual example of an autoencoding architecture.

If we divide the autoencoding structure as described above, we firstly obtain the encoder as we can see in figure 2.2 or formally defined as follows.

Definition 2.1.1. Let Θ be a parameter space and $\theta \in \Theta$ a set of parameters, $L \in \mathbb{N}$ and $d_1, \ldots, d_L \in \mathbb{N}$. Let further φ be an activation function and $f_{\varphi,L,\theta}$ a neural network.
If the neural network $f_{\varphi,L,\theta}$ fulfils the condition $n_i = d_1 \geq \ldots \geq d_L = n_o$ with $n_i, n_o \in \mathbb{N}$ being the input and output dimensions respectively, then we speak of an **encoding neural network** (or short: **encoder**) and denote it as f_e.

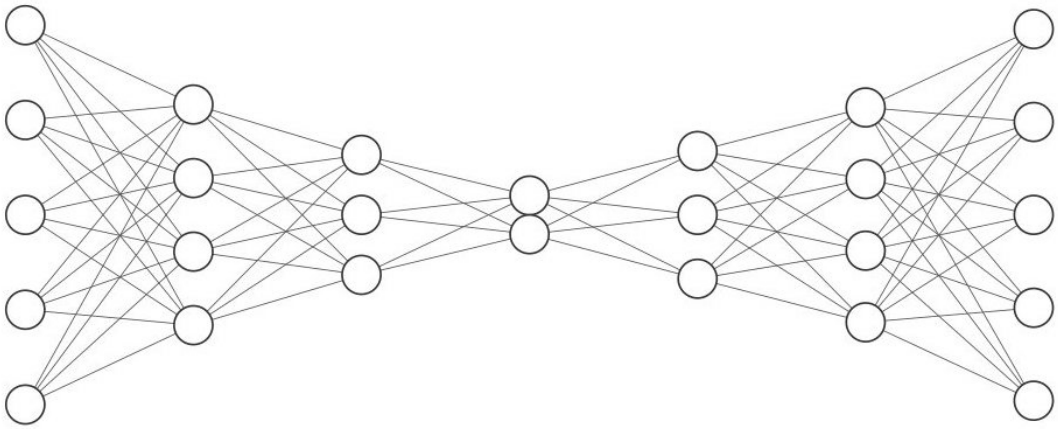

Figure 2.1: An autoencoding neural network with input and output $x, y \in \mathbb{R}^5$. The five hidden layers have dimensions 4, 3, 2, 3 and 4 respectively. Hence, the bottleneck dimension is 2 in this example. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

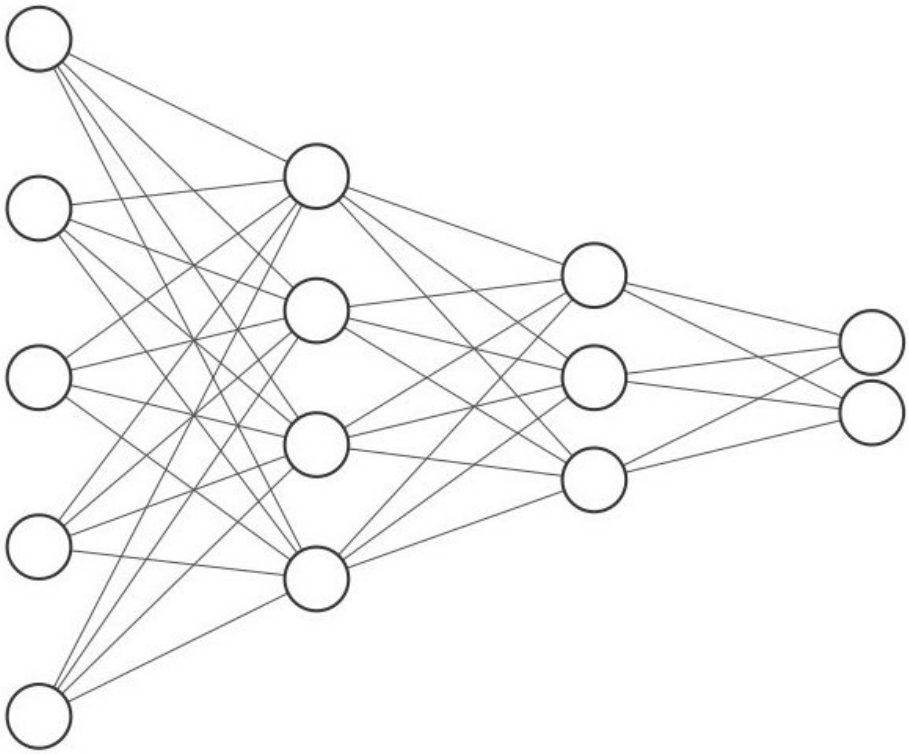

Figure 2.2: An encoding neural network with input $x \in \mathbb{R}^5$ and output $y \in \mathbb{R}^2$. The two hidden layers have dimensions 4 and 3. Hence, the encoder reduces the data dimensionality from 5 to 2 dimension. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

For the second part of the divided autoencoding structure, we obtain the decoder as we can see in figure 2.3. We can define this architecture analogously to the encoder in definition 2.1.1.

Definition 2.1.2. Let Θ be a parameter space and $\theta \in \Theta$ a parameter, $L \in \mathbb{N}$ and $d_1, \ldots, d_L \in$

$\mathbb{N}$. Let further φ be an activation function and $f_{\varphi,L,\theta}$ a neural network.
If the neural network $f_{\varphi,L,\theta}$ fulfils the condition $n_i = d_1 \leq \ldots \leq d_L = n_o$ with $n_i, n_o \in \mathbb{N}$ being the input and output dimensions respectively, then we speak of an **decoding neural network** (or short: **decoder**).

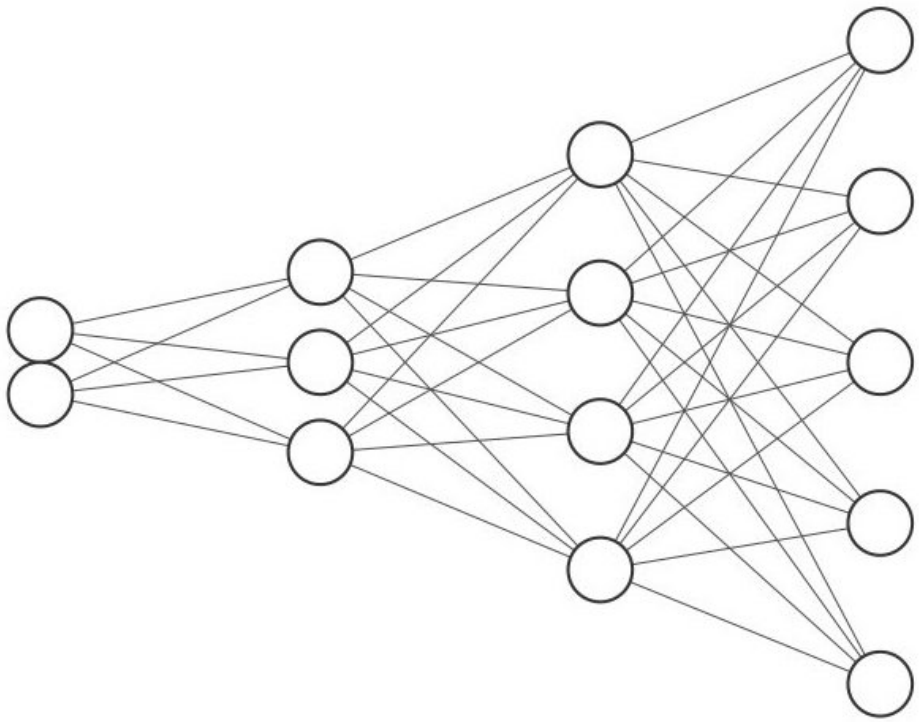

Figure 2.3: A decoding neural network with input $x \in \mathbb{R}^2$ and output $y \in \mathbb{R}^5$. The two hidden layers have dimensions 3 and 4. Hence, the decoder expands the data dimensionality from 2 to 5 dimensions. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

Before combining the encoding and the decoding structure to obtain the autoencoding neural network, we need to consider the following technicality first.

Lemma 2.1.3. *Let f_1, f_2 be two neural networks of depths $L_1, L_2 \in \mathbb{N}$ with parameters $\theta_1, \theta_2 \in \Theta$, where Θ is an arbitrary parameter space. Furthermore, let the dimensions of each layer be $d_1, \ldots, d_{L_1} \in \mathbb{N}$ of f_1 and $\widetilde{d}_1, \ldots, \widetilde{d}_{L_2} \in \mathbb{N}$ of f_2. Additionally, let $d_{L_1} = \widetilde{d}_1$.*
Then their composition $f_2 \circ f_1$ is a neural network of depth $L_1 + L_2$ with parameters (θ_1, θ_2).

Proof. Since f_1 is a neural network of depth L_1 with parameters θ_1, its architecture looks like

$$f_1(x) = H_{L_1} \circ H_{L_1-1} \circ \ldots H_2 \circ H_1(x), \qquad x \in \mathbb{R}^{d_1}. \tag{2.1}$$

Analogously, we can write f_2 as

$$f_2(y) = \widetilde{H}_{L_2} \circ \widetilde{H}_{L_2-1} \circ \ldots \widetilde{H}_2 \circ \widetilde{H}_1(y), \qquad y \in \mathbb{R}^{\widetilde{d}_1}. \tag{2.2}$$

Since we assumed that the output dimension d_{L_1} of the neural network f_1 is equal to the input dimension $\widetilde{d}_1$ of the neural network f_2, we can consider the result of (2.1) as input for (2.2)

$$y \coloneqq H_{L_1} \circ H_{L_1-1} \circ \ldots H_2 \circ H_1(x), \qquad x \in \mathbb{R}^{d_1}.$$

Hence, we obtain

$$\begin{aligned} f_2(y) &= \widetilde{H}_{L_2} \circ \widetilde{H}_{L_2-1} \circ \ldots \widetilde{H}_2 \circ \widetilde{H}_1(y), \\ f_2\left(f_1\left(x\right)\right) &= \widetilde{H}_{L_2} \circ \widetilde{H}_{L_2-1} \circ \ldots \widetilde{H}_2 \circ \widetilde{H}_1\left(H_{L_1} \circ H_{L_1-1} \circ \ldots H_2 \circ H_1(x)\right), \\ f_2\left(f_1\left(x\right)\right) &= \widetilde{H}_{L_2} \circ \widetilde{H}_{L_2-1} \circ \ldots \widetilde{H}_2 \circ \widetilde{H}_1 \circ H_{L_1} \circ H_{L_1-1} \circ \ldots H_2 \circ H_1(x), \qquad x \in \mathbb{R}^{d_1}. \end{aligned} \tag{2.3}$$

Therefore, from (2.3) follows that the composition $f_2 \circ f_1$ is a neural network of depth $L_1 + L_2$. Lastly, we consider the parameters θ of the neural network $f_2 \circ f_1$. Since the parameters of a neural network were defined as $\theta = (\theta_1, \ldots, \theta_L)$, where each entry is defined as $\theta_i = (W_i, b_i)$ and denotes the weight and bias of each layer H_i or $\widetilde{H}_i$, respectively, we can write the parameters of both neural networks as

$$\begin{aligned} \theta_1 &\coloneqq (\theta_1, \ldots, \theta_{L_1}) = \left((W_1, b_1), \ldots, (W_{L_1}, b_{L_1})\right), \\ \theta_2 &\coloneqq (\widetilde{\theta}_1, \ldots, \widetilde{\theta}_{L_2}) = \left((\widetilde{W}_1, \widetilde{b}_1), \ldots, (\widetilde{W}_{L_2}, \widetilde{b}_{L_2})\right). \end{aligned}$$

Hence, the composition $f_2 \circ f_1$ has the parameters

$$\begin{aligned} \theta &= \left((W_1, b_1), \ldots, (W_{L_1}, b_{L_1}), (\widetilde{W}_1, \widetilde{b}_1), \ldots, (\widetilde{W}_{L_2}, \widetilde{b}_{L_2})\right) \\ &= \left(\theta_1, \ldots \theta_{L_1}, \widetilde{\theta}_1, \ldots, \widetilde{\theta}_{L_2}\right) =: (\theta_1, \theta_2). \end{aligned}$$

□

Lemma 2.1.3 allows us to consider a modern approach to neural networks, a so called modular approach. Essentially, we consider entire structures like the encoding and the decoding neural network as a self-contained module. These modules can now easily be put together by considering them in series. This is very useful in practice, since modern neural networks consist of thousands of layers and thus billions of parameters. Considering a modular approach one can therefore divide the whole neural network and tune each module separately.

Theorem 2.1.4. *Let Θ be a parameter space, $N \in \mathbb{N}$ and $L_1, \ldots, L_N \in \mathbb{N}$. Furthermore, let $f_1, \ldots, f_N$ be neural networks with parameters $\theta_1, \ldots, \theta_N \in \Theta$ and depths $L_1, \ldots, L_N$, respectively. Lastly, let the output dimension of f_i match the input dimension of f_{i+1} for all $i \in \{1, \ldots, N-1\}$.*
Then the composition

$$f \coloneqq f_N \circ f_{N-1} \circ \ldots \circ f_1,$$

is a neural network with parameters $\theta = (\theta_1, \ldots \theta_N)$ of depth $L = L_1 + \ldots + L_N$.

Proof. Applying lemma 2.1.3 to f_1 and f_2 yields the composed neural network $f^{(1)} \coloneqq f_2 \circ f_1$ with parameters $\theta^{(1)} \coloneqq (\theta_1, \theta_2)$ and depth $L^{(1)} \coloneqq L_1 + L_2$.
If we now apply lemma 2.1.3 once again to $f^{(1)}$ and f_3, we receive $f^{(2)} \coloneqq f_3 \circ f^{(1)}$ with parameters $\theta^{(2)} \coloneqq (\theta^{(1)}, \theta_3) = (\theta_1, \theta_2, \theta_3)$ and depth $L^{(2)} \coloneqq L^{(1)} + L_3 = L_1 + L_2 + L_3$.
We realize, that iteratively applying lemma 2.1.3 yields after $N-1$ applications

$$\begin{aligned} f^{(N-1)} &= f_N \circ f_{N-1} \circ \ldots \circ f_1, \\ \theta^{(N-1)} &= (\theta_1, \ldots, \theta_N), \\ L^{(N-1)} &= \sum_{i=1}^{N-1} L_i. \end{aligned}$$

Therefore the assertion is proven. □

Definition 2.1.5. Let f_e and f_d be an encoding and a decoding neural network with input dimension n_i in $\mathbb{N}$ and output of the encoding neural network n_b in $\mathbb{N}$, that we will refer to as **bottleneck** of the autoencoding neural network. Then we define an **autoencoding neural network** f_a as the composition

$$\begin{aligned} f_a : \mathbb{R}^{n_i} &\to \mathbb{R}^{n_i}, \\ x &\mapsto (f_d \circ f_e)(x). \end{aligned}$$

Remark 2.1.6. Let f_e and f_d be an encoding and a decoding neural network as in definition 2.1.5. Then the composition $f_d \circ f_e$ is indeed a neural network, following from lemma 2.1.3.

2.2 Training of autoencoders

When tackling the question of how to train an autoencoding neural network, we realize that in contrast to regular neural networks, where we compare the output of the neural network to a label, in the current setting we can compare the input data to the computed output, since the goal of an autoencoding neural network ultimately is to alter and reconstruct images. In other words, we approach this optimization problem in an unsupervised learning setting. This forces us to consider slightly different loss functions than in the supervised learning setting.

Definition 2.2.1. Let X, Y be Banach spaces representing the input and output space, respectively.
Then a continuous function defined as

$$\begin{aligned} L : X \times Y &\to \mathbb{R}, \\ (x, p(x)) &\mapsto L(x, p(x)), \end{aligned}$$

is called **unsupervised loss function**.

There are multiple important loss functions in computer vision. We will consider a couple of those in the following example. For further reading please take a look at [3]

Example 2.2.2. Let Ω be a pixel domain with resolution (M, N) and d the number of channels. Furthermore, let f be a neural network with arbitrary but fixed architecture.
Then the following functions are loss functions operating on images.

Mean Squared Error (MSE):

$$\mathrm{L}_{\mathrm{MSE}}(\psi, f(\psi)) = \sum_{i=1}^{M} \sum_{j=1}^{N} \left\| \psi_{ij} - f(\psi)_{ij} \right\|^2,$$

Binary Cross-Entropy (BCE):

$$\begin{aligned} \mathrm{L}_{\mathrm{BCE}}(\psi, f(\psi)) = -\frac{1}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} \Big(& \psi_{ij} \log\left(f(\psi)_{ij} \right) \\ & + (1 - \psi_{ij}) \log\left(1 - f(\psi)_{ij} \right) \Big), \end{aligned}$$

where ψ denotes an image defined on Ψ_Ω.

Remark 2.2.3. The Binary Cross-Entropy loss function is usually used for binary classification problems. However, it still works in computer vision.

2.3 Applications

In this section we want to train a couple of own autoencoding neural networks on the MNIST dataset - a dataset consisting of handwritten digits. Lastly, we will consider a composition of MNIST images in order to simulate a scenario that will allow the Duckeneers GmbH to generate new data. We will consider some various architectures of neural networks and visualise the results in a comprehensible manner.
Firstly, we want to consider the most simple architecture, a fully connected linear neural network.

Definition 2.3.1. Let Θ be an arbitrary parameter space, $L \in \mathbb{N}$ and $d_1, \dots, d_L \in \mathbb{N}$. Furthermore, let φ be an arbitrary activation function.
Then an encoding neural network, where each layer $H_1, \dots, H_L$ is defined as

$$H_i(x) = \widehat{\varphi}(W_i x + b_i), \qquad x \in \mathbb{R}^{d_i}, i \in \{1, \dots, L\},$$

where $\theta_i(W_i, b_i) \in \Theta$ are the parameters of the i-th layer, see (1.2). Such a decoding neural network is called a **linear encoder**.

Analogously, we define a linear decoding neural network.

Definition 2.3.2. Let Θ be an arbitrary parameter space, $L \in \mathbb{N}$ and $d_1, \dots, d_L \in \mathbb{N}$. Furthermore, let φ be an arbitrary activation function.
Then a decoding neural network, where each layer $H_1, \dots, H_L$ is defined as

$$H_i(x) = \widehat{\varphi}(W_i x + b_i), \qquad x \in \mathbb{R}^{d_i}, i \in \{1, \dots, L\},$$

where $\theta_i = (W_i, b_i) \in \Theta$ are the parameters of the i-th layer H_i, see (1.2). Such a decoding neural network is called a **linear decoder**.

With the definitions of the linear encoder and linear decoder, we now can define a linear autoencoder.

Definition 2.3.3. Let f_e and f_d be a linear encoder and a linear decoder. Then a **linear autoencoder** f_{lin} is defined as the composition

$$f_{\text{lin}} \coloneqq f_d \circ f_e.$$

Before considering specific examples on the MNIST dataset, we need to consider some properties of the said dataset first.

Proposition 2.3.4. *The MNIST dataset $\mathcal{D}$ consists of greyscale images with a resolution of $(28, 28)$. Hence, the pixel domain Ω of $\mathcal{D}$ is $\Omega = \{1, \dots, 28\} \times \{1, \dots, 28\}$.*

Now, let us take a look at a specific example of a linear autoencoder on the MNIST dataset.

Algorithm 2 Linear autoencoder

Let the input and output dimensions be $n_i, n_o = 784$. Furthermore, let the encoder and the decoder have three hidden layers with dimensions $n_1 = 128, n_2 = 64$ and $n_3 = 12$ with bottleneck $n_b = 3$.
Let the chosen optimizer be ADAM with a learning rate of $\gamma = 3e - 4$ and the MSE loss function. We will perform the training on 10.000 epochs with a batch size of 512.

```
epochs ← 1000
epoch ← 0
for epoch ≤ epochs do
    for image in batch do
        image = image.reshape(784)          ▷ Convert the image from matrix to array.
        reconstructed = autoencoder(image)  ▷ Feed the image into the autoencoder.
        loss = MSE(reconstructed, image)    ▷ Compare the output to the input.
        backpropagation(loss)               ▷ Perform an optimizer step.
    end for
    epoch = epoch +1
end for
```

3 Variational Autoencoders

Differently from ordinary autoencoding neural networks, that we already mentioned to be discriminative models variational autoencodinf neural networks on the other hand are so called generative models. Instead of trying to estimate the conditional distribution of $y|x$, where y is a predicted label to an observation x, variational autoencoders attempt to capture the entire probability distribution of Y. These models are usually referred to as generative models, see [4, chapter 5] This is very interesting for multiple reasons, since this means that we can simulate and anticipate the evolution of the model output. Hence, we could generate new data based on the captured probability distribution. This will be our ultimate goal in this chapter, considering applications at last.

3.1 What are variational autoencoders?

Firstly, we want to define the specific structure of variational autoencoding neural networks and their key differences to ordinary autoencoding neural networks. As already motivated, variational autoencoders are generative models. This means, that they intend to capture the entire probability distribution of the observation space Y. Hence, they are statistical models and depend on certain probabilistic approaches.

In order to compute the gradient of a variational autoencoding neural network, we need to determine a way to quantify the distance between probability distributions. A popular measure for this case is the so called Kullback-Leibler divergence. It assesses the dissimilarity between two probability distributions over the same random variable X.

Definition 3.1.1. Let X be a random variable and p, q two probability density functions over X. Then, the **Kullback-Leibler divergence** is defined as

$$D_{\mathrm{KL}}(p \parallel q) = \int p(x) \log \left(\frac{p(x)}{q(x)} \right) dx. \tag{3.1}$$

Definition 3.1.2. Let X be a random variable and p a probability density function over X. Then, the nonnegative measure of the expected information content of X under correct distribution p, defined as

$$\mathcal{H}(X) = -\int p(x) \log p(x) dx = \mathbb{E}_p \left[-\log p(x) \right], \tag{3.2}$$

is called **entropy** of p.

Definition 3.1.3. Let X be a random variable and p, q two probability density functions over X. Then, the nonnegative measure of the expected information content of X under incorrect

distribution q, defined as

$$\mathcal{H}_q(X) = -\int p(x) \log q(x) dx = \mathbb{E}_p \left[-\log q(x) \right], \tag{3.3}$$

is called **cross-entropy** between p and q.

Remark 3.1.4. Let X be a random variable and p, q probability density functions over X. Then the Kullback-Leibler divergence between p and q can be written as follows

$$D_{\mathrm{KL}}(p \parallel q) = \mathcal{H}_q(p) - \mathcal{H}(p),$$

where $\mathcal{H}_q$ and $\mathcal{H}$ denote the cross-entropy and entropy, respectively.

Bibliography

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[2] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[3] D. Foster, *Generative deep learning*. " O'Reilly Media, Inc.", 2022.

[4] L. P. Cinelli, M. A. Marins, E. A. B. Da Silva, and S. L. Netto, *Variational methods for machine learning with applications to deep networks*. Springer, 2021.

Stuttgart, August 21, 2023