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

TODO: Here should be a catchy abstract and maybe even a short introduction.

Contents

1 Preliminary

In order to understand 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 - 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 is a zoo 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: $$\varphi(t) = \frac{1}{1+e^{-x}}.$$

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 generalize the φ-layer $H : \mathbb{R}^k \to \mathbb{R}^m$ by

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

Remark 1.1.5. Let the same assumptions as in definition 1.1.4 hold.
From now on we want to summarize 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 1.1.6. Let $\varphi : \mathbb{R} \to \mathbb{R}$ be an activation function and $H_1, \dots, H_L$ with $L \in \mathbb{N}$ be φ-layers with parameters $\theta_i = (W_i, b_i)$. Then, with $\theta = (\theta_1, \dots \theta_L)$ the function $f_{\varphi,L,\theta} : \mathbb{R}^{d_1} \to \mathbb{R}^{d_L}$ defined by

$$f_{\varphi,L,\theta}(x) \coloneqq H_L \circ \dots \circ H_1(x), \qquad x \in \mathbb{R}^{d_1}, \tag{1.3}$$

is called a **φ-deep neural network** of depth L with parameters $\theta \in \Theta$, where d_1 describes the input dimension and d_L the output dimension respectively and Θ is some arbitrary parameter space.
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.

A visual representation of a neural network can be found in figure 1.1

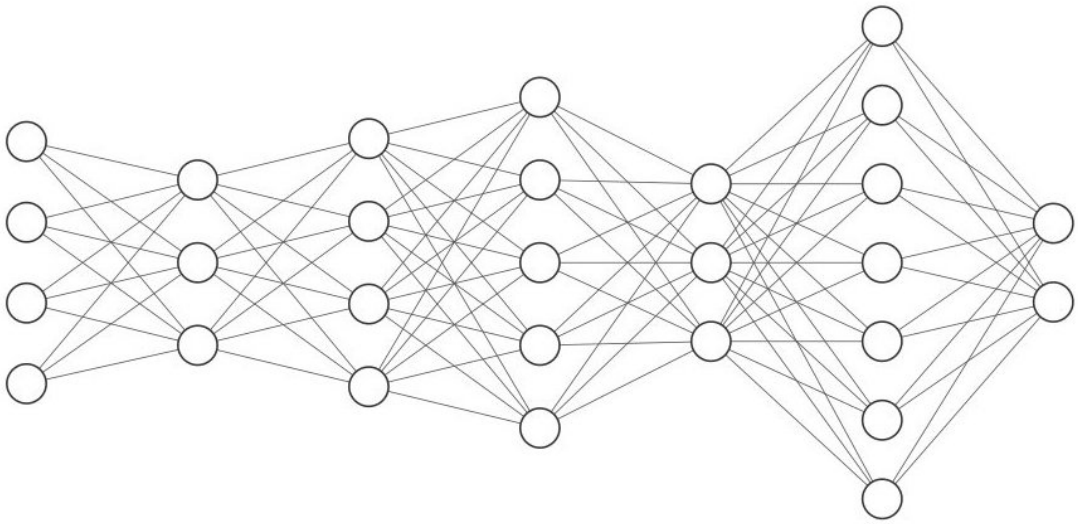

Figure 1.1: A neural network with input $x \in \mathbb{R}^4$ and output $y \in \mathbb{R}^2$. The five hidden layers have dimensions 3, 4, 5, 3 and 7 respectively. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

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 $Y \subseteq \mathbb{R}^n$ be arbitrary Banach spaces, that we will refer to as input and output space, $p : X \to \mathbb{R}^n$ be a continuous, convex function.
Furthermore, let $L : X \times Y \times \mathbb{R}^n \to [0, \infty)$ be a **supervised loss function**, a measurable function that compares a true value $y \in Y \subset \mathbb{R}^n$ to a predicted value $\widehat{y} = p(x)$.
Lastly, let P be a probability measure on $X \times Y$. Then the **L-risk function** $\mathcal{R} : X \times Y \times \mathbb{R}^n \to [0, \infty)$ with regard to a loss function L is defined as

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

One may have already noticed, that the definition of a loss function is very general. This flexibility is quite important, since in different use cases the characteristics of the problem could differ immensely: are large errors just as bad as small ones or considerably worse?
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. The following functions are loss functions.

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

Linear Error Loss: $L(x, y, p(x)) = \|y - p(x)\|$,

There still are other important loss functions that we will consider in chapter 3.
Considering applications, one usually wants to compute the risk with regard to some observed data instead of some unknown probability measure. In this case the general risk function becomes more tangible, as we see in the following definition.

Definition 1.2.3. Let $X \subseteq \mathbb{R}^d$ and $Y \subseteq \mathbb{R}^n$ be arbitrary Banach spaces and $d, n \in \mathbb{N}$, $D = ((x_1, y_1), \ldots, (x_k, y_k))$ be a dataset consisting of $k \in \mathbb{N}$ data points. Furthermore, let L be a loss function and p be an arbitrary prediction function as in definition 1.2.1.
Then we define the **empirical risk function** as

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

Since we will mostly consider the practical setting where we have a dataset given, 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 1.2.4. *Let $(\gamma_t)_{t\in\mathbb{N}}$ be a converging sequence of step sizes with $\gamma_t \to 0$. Let further be $f : \mathbb{R}^n \to \mathbb{R}$ 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 $x' := \frac{x_1^*+x_2^*}{2}$, 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. □

Since we are considering neural networks in this thesis, we want to take a quick look on how we can apply theorem 1.2.4 to a neural network. But in order to do so, 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 1.2.5. *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. Furthermore, let D be a dataset of length $k \in \mathbb{N}$ and L be an arbitrary loss function as in definition 1.2.1.*
The gradient of the risk function $\mathcal{R}(\cdot)$ with regard to the neural network f_θ and thus the parameters θ looks 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 1.2.3 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 now can formulate the gradient descent algorithm for a neural network.

Corollary 1.2.6. *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 of step sizes 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 1.2.4. 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 1.2.5

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 1.2.7. *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 as previous 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
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.

Definition 1.3.1. Let $M, N \in \mathbb{N}$ and $\Omega = \{1, \dots, M\} \times \{1, \dots, N\} \subset \mathbb{N}^2$ be called **pixel domain** of an image with the tuple (M, N) being called **resolution**. Then we define a **digital image** by $\psi_{ij} \in \Psi_d \coloneqq \{1, \dots, 255\}^d$ for all (i, j) in Ω, where $d \in \mathbb{N}$ represents the number of **channels** in the picture.
We will write $\psi \in \Psi_{d,\Omega} \coloneqq \Psi_d^{M\times N}$ from now on.

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

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 those. But first, we need some technical assertions.

Lemma 1.3.3. *Let Ω be a pixel domain with resolution (M, N). Then the picture 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 a picture 1.3.1. Hence, ψ is a $M \times N$- 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

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

□

Lemma 1.3.3 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 1.3.4. 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 1.3.4 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 1.3.5. *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 1.3.4, b is a bias and Ω_1 a pixel domain with resolution (M_1, N_1).

With the help of corollary 1.3.5 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 1.3.6. *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 1.3.7. *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 1.3.8. *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** $T_{min\text{-}pool} : \Psi_{\Omega_0} \to \Psi_{\Omega_1}$ with stride s and pooling I is defined by*

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

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 1.3.9. *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.
One should mention, that since the goal of an autoencoding neural network ultimately is to alter and reconstruct images, we approach this in an unsupervised learning setting. Hence, we compare the output of the neural network to the input. This forces us to consider slightly different loss functions.

Definition 2.1.1. 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 the computer vision setting. We will consider a couple of those in the following example. For further reading please take a look at [2]

Example 2.1.2. 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):

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

where ψ denotes an image defined on Ψ_Ω.

Remark 2.1.3. The Binary Cross-Entropy loss function is usually used for binary classification problems. However, it still works in computer vision.

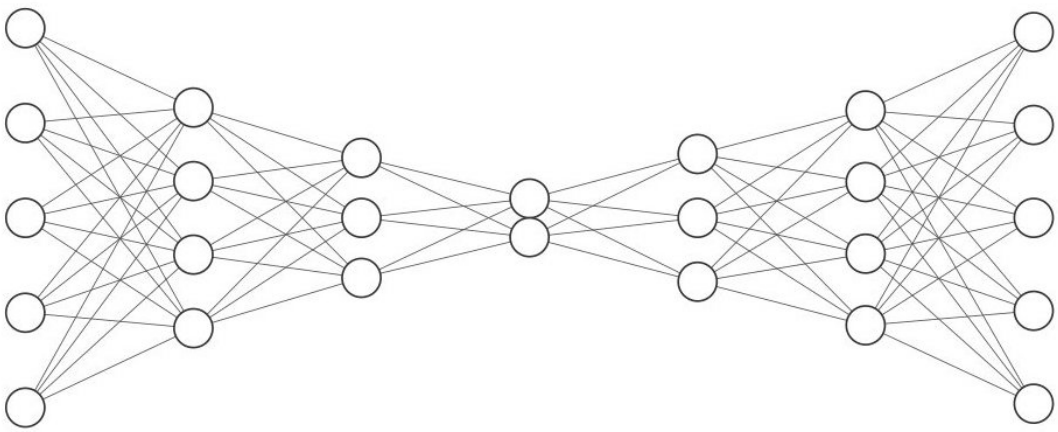

Figure 2.1: An autoencoding neural network with input and output $x, y \in \mathbb{R}^5$. The five hidden layers have dimensions 4, 3, 2, 3 and 4 respectively. Hence, the bottleneck dimension is 2 in this example. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

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

Definition 2.1.4. 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**).

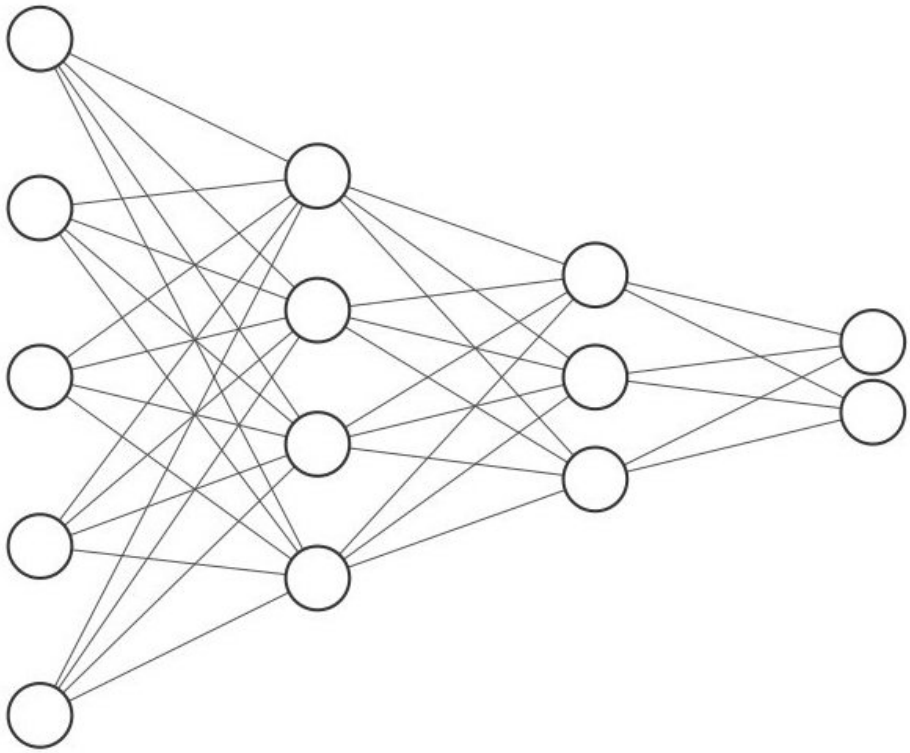

Figure 2.2: An encoding neural network with input $x \in \mathbb{R}^5$ and output $y \in \mathbb{R}^2$. The two hidden layers have dimensions 4 and 3. Hence, the encoder reduces the data dimensionality from 5 to 2 dimension. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

For the second part of the divided autoencoder structure, we obtain the decoder as we can see in figure 2.3. We can define this architecture analogously to the encoder in lemma 2.1.4.

Definition 2.1.5. 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**).

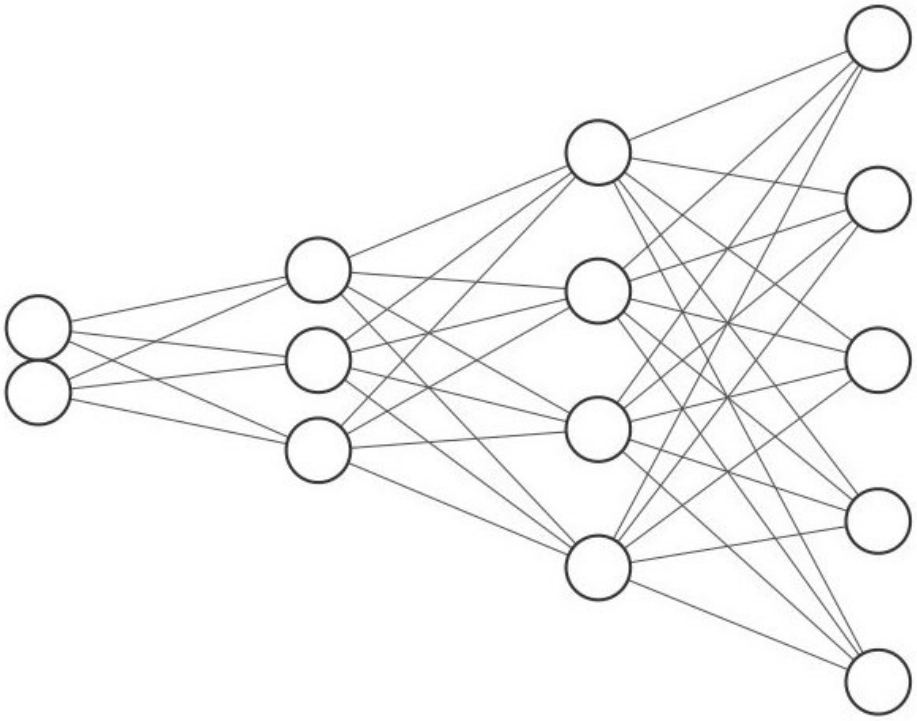

Figure 2.3: A decoding neural network with input $x \in \mathbb{R}^2$ and output $y \in \mathbb{R}^5$. The two hidden layers have dimensions 3 and 4. Hence, the decoder expands the data dimensionality from 2 to 5 dimensions. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

3 Variational Autoencoders

Differently form autoencoders, that we already mentioned to be discriminative models variational autoencoders 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 [3, 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 application at least.

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 3.0.1. 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 3.0.2. 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 3.0.3. 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 3.0.4. 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. Foster, *Generative deep learning*. " O'Reilly Media, Inc.", 2022.

[3] 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 1, 2023