Master's thesis

June 26, 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 further, we will consider a couple of useful operations that neural networks are capable of doing. Lastly, we will take a look at some strategies of training neural networks.

1.1 Neural networks

Originally, the idea of neural networks came 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. Scientist tried to mimic this natural architecture and replicate the human 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. But what do these neural networks do? Why are they so popular? What actually is a neural network? All those are very interesting and important questions that we will find answers for.
As already mentioned, neural networks consist of single neurons that move around information upon being „triggered". Obviously, triggering an artificial neuron can't happen the same way as neurological neurons are being triggered. 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.

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

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.

In order to expand the architecture, we consider multiple neurons being arranged in a so called layer.

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 $\hat{\varphi} : \mathbb{R}^k \to \mathbb{R}$ as the component-wise mapping of $\varphi : \mathbb{R} \to \mathbb{R}$, meaning $\hat{\varphi}(v) = (\varphi(v_1), \dots, \varphi(v_k))$, we can generalize the φ-layer $H : \mathbb{R}^k \to \mathbb{R}^m$ by

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

Finally, we can introduce neural networks with the previous definitions formally.

Definition 1.1.5. 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 $\theta_i = (W_i, b_i)$ as in definition 1.1.4. 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

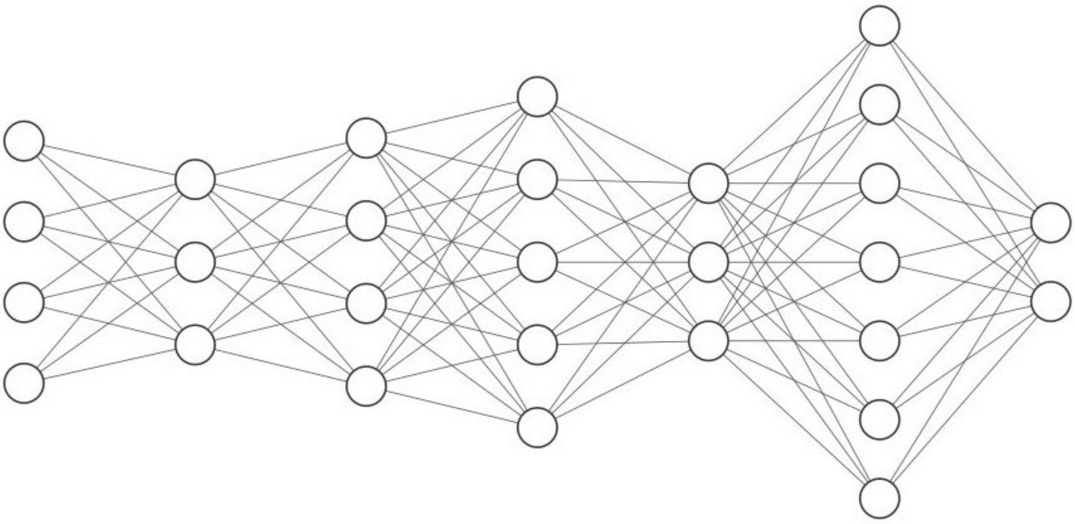

Figure 1.1: A neural network with input $x \in \mathbb{R}^4$ and output $y \in \mathbb{R}^2$. The five hidden layers have dimensions 3, 4, 5, 3 and 7 respectively. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

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 called training of a neural network. There are many approaches of how to train a neural network. However, most of them rely on iteratively finding the gradient - the direction of greatest ascent. In the following we want to consider a couple of popular algorithms that are used to train neural networks.

Theorem 1.2.1. *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*

$$x^{(t+1)} = x^{(t)} - \gamma_t \, \partial_x f(x^{(t)}), \tag{1.4}$$

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

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

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 1.2.1 to a neural network. But firstly, we need to define the so called loss function and risk function - functions to measure the error of a neural network, or any prediction function in general. This is fundamental in supervised learning.

Definition 1.2.2. Let $X \subseteq \mathbb{R}^d$ and $Y \subseteq \mathbb{R}^n$ be arbitrary Banach spaces and $d, n \in \mathbb{N}$, 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 **loss function**, a measurable function that compares a true value $y \in Y \subset \mathbb{R}^n$ to a predicted value $\hat{y} = p(x)$.
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

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

In applications one usually wants to compute the risk with regard to some observed data. In this case the general definition of a 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 1.2.2.
Then we define the **empirical risk function** as

$$\mathcal{R}_{L,D}(p) = \frac{1}{k} \sum_{i=1}^{k} L\left(x_i, y_i, p(x_i)\right). \tag{1.5}$$

Since we will mostly consider the practical setting where we have a dataset given, we will write $\mathcal{R} := \mathcal{R}_{L,D}$ unless unclear in the given context.

With the above definitions we now need to consider one last thing in order to formulate the gradient descent algorithm for neural networks. This last thing is the question how actually to compute the gradient of a neural network.

Lemma 1.2.4. *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 1.2.2.*
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 above definitions we now can formulate the gradient descent algorithm for a neural network.

Corollary 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. Let $(\gamma_t)_{t\in\mathbb{N}}$ be a converging 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 1.2.1. 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 1.2.4

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 costly to compute the gradient, 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 following.

Theorem 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 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 come pages to introduce.. Is it worth it?

2 Autoencoders

Now, having introduced the basics of neural networks in Chapter 1 we can consider a specific architecture of a neural network, a so called autoencoder neural network, or short: autoencoders. The conceptional idea of autoencoders is to take a given input, compress (usually called encore) the input to a given size and afterwards, expand (usually called decode) it as close as possible to the original representation again. Such an architecture is widely used in different areas. 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 being encoded first 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 autoencoders, since it is way more efficient to first encode the data and then fit the 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 firstly, 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.
The conceptional idea of autoencoders is now clear, but how exactly would one formulate such an architecture mathematically? This is the central question we want to answer in this chapter.

2.1 Mathematical formulation of autoencoders

As already mentioned, the input data is firstly being encoded, and afterwards it is being decoded. Hence, we can divide these two steps into separate architectures - the encoder and the decoder, which we will formulate separately. In figure 2.1 we can take a look at a visual example of an autoencoder architecture.

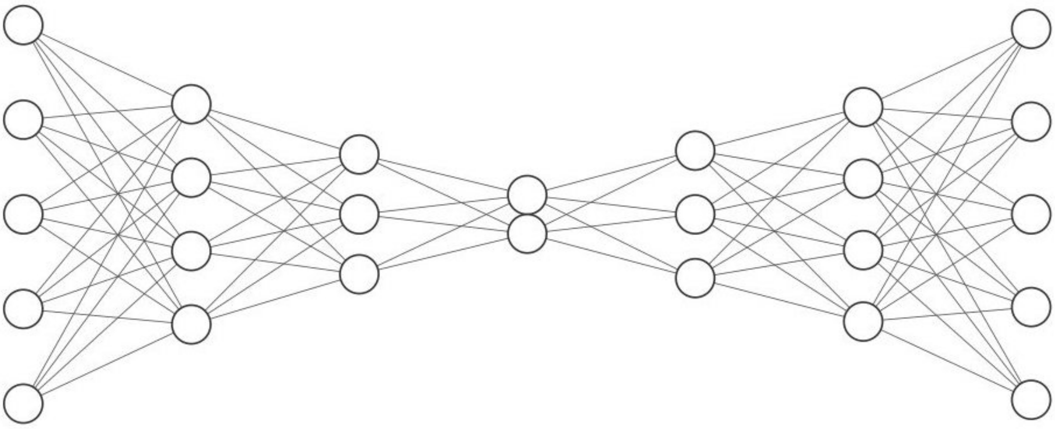

Figure 2.1: An autoencoder neural network with input and output $x, y \in \mathbb{R}^5$. The five hidden layers have dimensions 4, 3, 2, 3 and 4 respectively. Hence, the bottleneck dimension is 2 in this example. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

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

Lemma 2.1.1. *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 $n_i = d_1 \geq \ldots, \geq d_L = n_o$ 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**).*

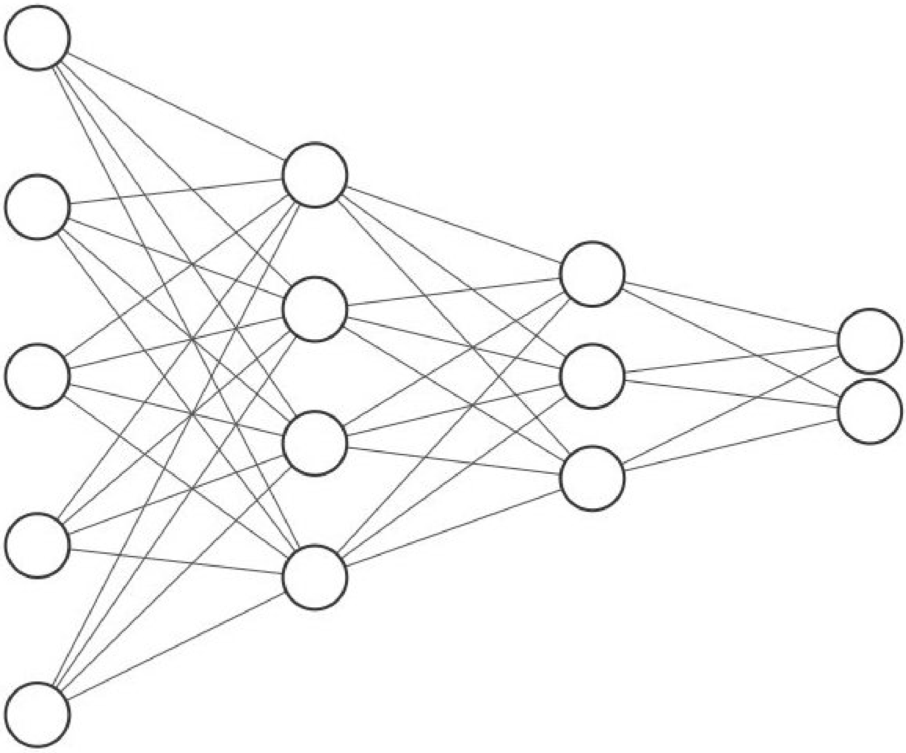

Figure 2.2: An encoder neural network with input $x \in \mathbb{R}^5$ and output $y \in \mathbb{R}^2$. The two hidden layers have dimensions 4 and 3. Hence, the encoder reduces the data dimensionality from 5 to 2 dimension. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

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 2.1.1.

Lemma 2.1.2. *Let Θ be a parameter space and $\theta \in \Theta$ a parameter, $L \in \mathbb{N}$ and $d_1, \dots, d_L \in \mathbb{N}$. 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 $n_i = d_1 \leq \dots, \leq d_L = n_o$ 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****).*

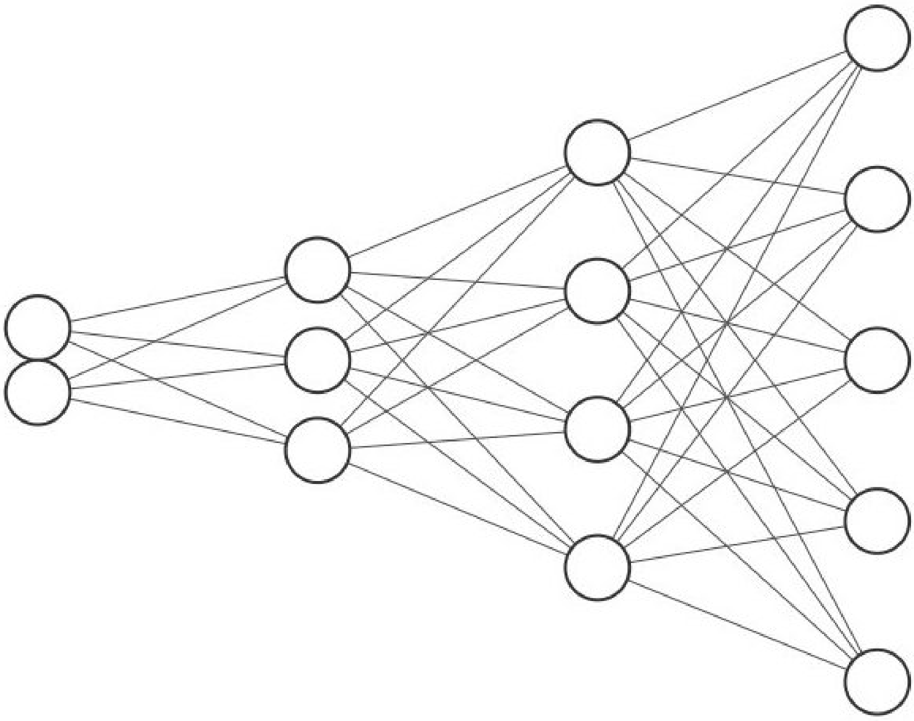

Figure 2.3: A decoder neural network with input $x \in \mathbb{R}^2$ and output $y \in \mathbb{R}^5$. The two hidden layers have dimensions 3 and 4. Hence, the decoder expands the data dimensionality from 2 to 5 dimensions. The graphic was generated with http://alexlenail.me/NN-SVG/index.html

Stuttgart, June 26, 2023