Assignment 4 (theoretical part)

Deadline: Friday, June 14th (23:59)

May 29, 2019

NOTE: The assignment consists of a theoretical and a practical part. This theoretical part contains an extensive model description, as well as some questions. Make sure you first read this part well and answer the questions here. Submit your answers in a separate PDF file, preferably created with LaTeX. Then, proceed with the practical part.

1 Variational Autoencoder (VAE)

In the practical part of this assignment, you will implement a variational autoencoder in Keras. But first we will go through the theoretical derivation of a VAE, as a guide for the implementation.

You are given a dataset $\mathcal{X} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}\}$ of dimensionality D, i.e. each data point $\boldsymbol{x} = (x_1, \ldots, x_D)^T$ is a D-dimensional vector. Let's assume this data set is sampled from a true (unknown) distribution $p(\boldsymbol{x})$, i.e. $\mathcal{X} \overset{iid}{\sim} p(\boldsymbol{x})$. To approximate this distribution, we will train a generative model $p_\theta(\boldsymbol{x})$ by means of a latent variable model $p_\theta(\boldsymbol{x}|\boldsymbol{z})$ that uses an approximate posterior distribution $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ to infer latent variables $\boldsymbol{z}$ given a data point $\boldsymbol{x}$. We will derive the model step by step, ending up at a variational autoencoder.

1.1 Maximum likelihood estimation

The model will be trained by means of maximum likelihood estimation, i.e. we will train it by maximising

$$\log p(\mathcal{X}) = \log \prod_{n=1}^{N} p_\theta(\boldsymbol{x}^{(n)}) = \sum_{n=1}^{N} \log p_\theta(\boldsymbol{x}^{(n)}), \tag{1}$$

where we can express the likelihood of a data point $\boldsymbol{x}$ with continuous latent variables $\boldsymbol{z}$:

$$\log p_\theta(\boldsymbol{x}) = \log \int p_\theta(\boldsymbol{x}, \boldsymbol{z}) d\boldsymbol{z} = \log \int p_\theta(\boldsymbol{x}|\boldsymbol{z}) p(\boldsymbol{z}) d\boldsymbol{z}. \tag{2}$$

Thus, we first need to define a prior distribution $p(\boldsymbol{z})$ from which we can sample latent variables, as well as a parametrised generative model $p_\theta(\boldsymbol{x}|\boldsymbol{z})$ from which we can sample new data points, given a latent variable.

Question 1 *(1pt) Given that the data is high-dimensional (in our case, images), what could be the motivation for training a latent variable model, as opposed to directly trying to model $p_\theta(\boldsymbol{x})$?*

1.2 Prior distribution

We will use a simple prior distribution for the latent variables; a standard factorised Gaussian (Normal) distribution:

$$p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{0}, I) \tag{3}$$
$$= \prod_{k=1}^{K} \mathcal{N}(z_k|0,1) \tag{4}$$
$$= \prod_{k=1}^{K} \frac{1}{\sqrt{2\pi}} e^{\frac{z_k^2}{2}}, \tag{5}$$

where $\boldsymbol{z} = (z_1, \ldots, z_K)^T$, and K is the number of latent dimensions.

1.3 Generative/conditional distribution(s)

To create an expressive generative model, we use neural networks to parametrise $p_\theta(\boldsymbol{x}|\boldsymbol{z})$. In particular we will consider two different models. For binary data (i.e. each data point $\boldsymbol{x}$ is a vector of 0's and 1's), we use a factorised Bernoulli distribution:

$$p(\boldsymbol{x}|\boldsymbol{z}) = \prod_{d=1}^{D} \text{Bernoulli}(x_d|\rho_d) \tag{6}$$
$$= \prod_{d=1}^{D} \rho_d^{x_d}(1-\rho_d)^{(1-x_d)}, \tag{7}$$

where $\boldsymbol{\rho} = (\rho_1, \ldots, \rho_D)^T = \boldsymbol{\rho}(\boldsymbol{z})$ is modelled with a neural network that takes latent variables $\boldsymbol{z}$ as inputs. Note that for each d we must have $0 \leq \rho_d \leq 1$ to obtain valid Bernoulli distributions.

Question 2 *(1pt) Write out* $\log p(\boldsymbol{x}|\boldsymbol{z})$ *for this discrete model, and simplify the expression as much as possible. Can you relate this expression to a commonly used loss function for neural networks?*

For continuous data (i.e. each data point $\boldsymbol{x}$ is a vector of real numbers), we use a factorised Gaussian (Normal) distribution:

$$p(\boldsymbol{x}|\boldsymbol{z}) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2))$$
$$= \prod_{d=1}^{D} \mathcal{N}(x_d|\mu_d, \sigma_d^2)$$
$$= \prod_{d=1}^{D} \frac{1}{\sqrt{2\pi\sigma_d^2}} e^{-\frac{(x_d-\mu_d)^2}{2\sigma_d^2}},$$

where $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_D)^T = \boldsymbol{\mu}(\boldsymbol{z})$ is modelled with a neural network that takes latent variables $\boldsymbol{z}$ as inputs. Similarly, we could learn $\boldsymbol{\sigma}$ with a neural network as well, but to simplify our model we choose a fixed value $\sigma_d = \sigma$ that is the same for all $d = 1, \ldots, D$.

Question 3 *(1pt) Write out* $\log p(\boldsymbol{x}|\boldsymbol{z})$ *for this continuous model, and simplify the expression as much as possible. Can you relate this expression to a commonly used loss function for neural networks? (Hint: note that terms that are constant w.r.t. the learned parameters* $\boldsymbol{\mu}$ *will not affect the learning, as their derivative will be zero.)*

1.4 Variational lower bound & Approximate posterior distribution

As mentioned before, we want to train the model with maximum likelihood estimation, which in our formulation means maximising $\log p_\theta(\boldsymbol{x}) = \log \int p_\theta(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})d\boldsymbol{z}$. We can use gradient *ascent* (not descent) for this, which requires computing the gradients $\nabla_\theta \log \int p_\theta(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})d\boldsymbol{z}$, but these gradients require an integration over a neural network, which is intractable (i.e. would take too long to compute in practice). Moreover, there is no way to efficiently estimate these gradients.

Instead, we will define an approximate posterior distribution (or variational distribution) $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ to infer latent variables given observed data points, which allows us to formulate the variational lower bound (or evidence lower bound, "ELBO") on the likelihood:

$$\log p_\theta(\boldsymbol{x}) \geq \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] - KL(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})) \tag{8}$$

$$=: \mathcal{L}(\boldsymbol{x};\theta,\phi). \tag{9}$$

For this ELBO, we will see that we can obtain tractable (estimates of) gradients, thus we will maximise the ELBO instead.

For our model, we choose a factorised Gaussian, where both the means and variances are parametrised by neural networks:

$$q(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{m}, \text{diag}(\boldsymbol{s}^2)) \tag{10}$$

$$= \prod_{k=1}^{K} \mathcal{N}(z_k|m_k, s_k^2) \tag{11}$$

$$= \prod_{k=1}^{K} \frac{1}{\sqrt{2\pi s_k^2}} e^{-\frac{(z_k-m_k)^2}{2s_k^2}}, \tag{12}$$

where $\boldsymbol{m} = (m_1, \ldots, m_D)^T = \boldsymbol{m}(\boldsymbol{x})$ and $\boldsymbol{s} = (s_1, \ldots, s_D)^T = \boldsymbol{s}(\boldsymbol{x})$ are modelled with a neural network that takes data points $\boldsymbol{x}$ as inputs. *Note: we use $\boldsymbol{m}$ and $\boldsymbol{s}$ here for the mean and variance, to avoid confusion with the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ from the continuous generative model $p(\boldsymbol{x}|\boldsymbol{z})$.*

With this model, the KL-divergence term in eq. (8) can be computed analytically as follows[1]:

$$KL(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})) = \frac{1}{2}\sum_{k=1}^{K}(m_k^2 + s_k^2 - \log s_k^2 - 1). \tag{13}$$

Gradients w.r.t. $\boldsymbol{m}$ and $\boldsymbol{s}$ can be computed exactly for this expression, so it can directly be used in a neural network loss function. In the next section we will see that this is not the case for the first term in the right-hand side of eq. (8) however.

1.5 Reparametrisation trick

If we want to use the (negative) ELBO as a loss function, we need to be able to obtain gradients of it w.r.t. the parameters we are trying to learn. Consider the first right-hand side term from eq. (8), which by definition of the expectation can be written as follows:

$$\mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] = \int q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log p_\theta(\boldsymbol{x}|\boldsymbol{z}) dz. \tag{14}$$

For the parameters θ of the generative distribution, we can compute gradients as follows:

$$\nabla_\theta \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] = \nabla_\theta \int q_\phi(\boldsymbol{z}|\boldsymbol{x}) \log p_\theta(\boldsymbol{x}|\boldsymbol{z}) dz \tag{15}$$

$$= \int q_\phi(\boldsymbol{z}|\boldsymbol{x}) \nabla_\theta \log p_\theta(\boldsymbol{x}|\boldsymbol{z}) dz \tag{16}$$

$$= \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\nabla_\theta \log p_\theta(\boldsymbol{x}|\boldsymbol{z})]. \tag{17}$$

[1] see e.g. `https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence#Multivariate_normal_distributions`

We obtain this by inverting the order of the gradient and integration[2], and then moving $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ outside of the gradient (this is possible since it doesn't depend on θ). Although the result still contains an integral over a neural network, and is therefore intractable, it can be written as an expectation. Expectations can be estimated efficiently by means of Monte Carlo sampling; we simply sample a number of times (say L times) from the distribution $q_\phi(\boldsymbol{z}|\boldsymbol{x})$[3], and compute the average over the quantity inside the expectation for all samples:

$$\boldsymbol{z}^{(l)} \sim q_\phi(\boldsymbol{z}|\boldsymbol{x}) \text{ for } l = 1, \ldots, L \tag{18}$$

$$\mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\nabla_\theta \log p_\theta(\boldsymbol{x}|\boldsymbol{z})] \approx \frac{1}{L} \sum_{l=1}^{L} \nabla_\theta \log p_\theta(\boldsymbol{x}|\boldsymbol{z}^{(l)}). \tag{19}$$

Since neural network training requires many iterations, it is common to use a single sample here, i.e. we choose $L = 1$. This turns out to work well enough already in practice.

For the parameters ϕ of the approximate posterior distribution, we cannot perform the same step from eq. (15) to eq. (16) for the gradients ∇_ϕ, since $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ depends on ϕ. Since we chose $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ to be a Gaussian however, we can perform the so-called *reparametrisation trick*. In particular, it is important that we can rewrite $\boldsymbol{z}$ as a deterministic function g over a sample from a parameterless distribution, by means of a location-scale transformation. I.e. if

$$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, I), \tag{20}$$

then

$$g_\phi(\boldsymbol{\epsilon}) \sim \mathcal{N}(\boldsymbol{m}, \text{diag}(\boldsymbol{s}^2)), \tag{21}$$

for

$$g_\phi(\boldsymbol{\epsilon}) := \boldsymbol{m} + \boldsymbol{s} \odot \boldsymbol{\epsilon}, \tag{22}$$

where $\odot$ represents the element-wise product, and $\boldsymbol{m}$ and $\boldsymbol{s}$ are neural network outputs as defined in eq. (10). This means that to sample $\boldsymbol{z}$, we can instead sample an $\boldsymbol{\epsilon}$ from a standard Gaussian distribution, and compute $\boldsymbol{z} = g_\phi(\boldsymbol{\epsilon})$. Note that here, $\phi = \{\boldsymbol{m}, \boldsymbol{s}\}$. But this also implies that we can rewrite eq. (14) as follows:

$$\mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] = \mathbb{E}_{\mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0},I)}[\log p_\theta(\boldsymbol{x}|g_\phi(\boldsymbol{\epsilon}))]. \tag{23}$$

The gradient w.r.t. ϕ can now be rewritten as an expectation:

$$\begin{aligned}
\nabla_\phi \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] &= \nabla_\phi \mathbb{E}_{\mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0},I)}[\log p_\theta(\boldsymbol{x}|g_\phi(\boldsymbol{\epsilon}))] && (24)\\
&= \nabla_\phi \int \mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0}, I) \log p_\theta(\boldsymbol{x}|g_\phi(\boldsymbol{\epsilon})) && (25)\\
&= \int \mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0}, I) \nabla_\phi \log p_\theta(\boldsymbol{x}|g_\phi(\boldsymbol{\epsilon})) && (26)\\
&= \mathbb{E}_{\mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0},I)}[\nabla_\phi \log p_\theta(\boldsymbol{x}|g_\phi(\boldsymbol{\epsilon}))], && (27)
\end{aligned}$$

so with this reparametrisation, we can again obtain Monte Carlo estimates of this gradient:

$$\boldsymbol{\epsilon}^{(l)} \sim \mathcal{N}(\mathbf{0}, I) \text{ for } l = 1, \ldots, L \tag{28}$$

$$\mathbb{E}_{\mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0},I)}[\nabla_\phi \log p_\theta(\boldsymbol{x}|g_\phi(\boldsymbol{\epsilon}))] \approx \frac{1}{L} \sum_{l=1}^{L} \nabla_\theta \log p_\theta(\boldsymbol{x}|g_\phi(\boldsymbol{\epsilon})). \tag{29}$$

Again, in practice we will use a single sample, i.e. $L = 1$.

[2] this is known as Leibniz's rule

[3] note that this can be done efficiently, we compute the parameters ϕ once with a forward pass through the neural network, and then sample from the now known distribution $q_\phi(\boldsymbol{z}|\boldsymbol{x})$

1.6 Autoencoder structure

Putting all components together, we can see that the model resembles an autoencoder (hence the name Variational Autoencoder). Figure 1 visualises the computational flow of our VAE model. Note that we model the logarithm of the variance of the approximate distribution, instead of the variance itself, for computational reasons. We can see that inferring the parameters of the approximate posterior $q(\boldsymbol{z}|\boldsymbol{x})$ from an input x can be interpreted as a kind of encoder, latent variables are then sampled (with the reparametrisation trick) from these parameters, and inferring the mean of the generative distribution can then be interpreted as a kind of decoder. The output $\boldsymbol{\mu}$ can be interpreted as a mean reconstruction of the input data (with some predefined variance σ^2 on each pixel).

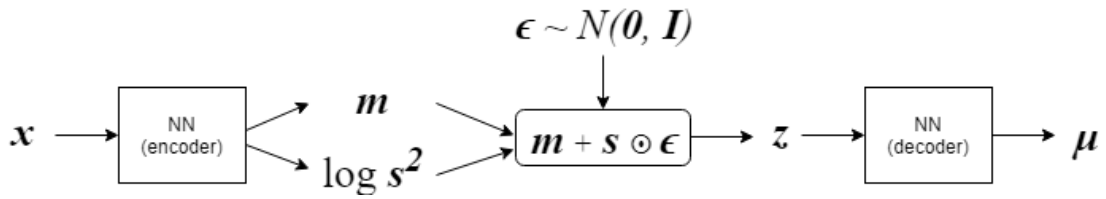

Figure 1: VAE overview

As explained before, we will train the network by maximising the ELBO with stochastic gradient *ascent* for this, which is equivalent to minimising the negative ELBO with stochastic gradient *descent*. The negative of the first term in the ELBO (see eqn. (8)) can then be interpreted as a kind of reconstruction loss between the input $\boldsymbol{x}$ and the output parameter $\boldsymbol{\mu}$. The KL divergence in the second term of the (negative) ELBO acts as a regulariser, preventing the approximate posterior from diverging too far from the prior.

The result is a model very similar to an autoencoder, but with a sampling process and extended loss function that have a clear mathematical justification.