Machine Learning for Signal Processing

[5LSL0]

Rik Vullings
Ruud van Sloun
Nishith Chennakeshava
Hans van Gorp

Assignment: (Variational) Autoencoders

April 2022

(Variational) Autoencoders

In this assignment you will implement and analyze an autoencoder and study the difference between deterministic and variational autoencoders.

For questions where you need to code in Python, provide the most relevant lines of code and show the plots generated with the code in your report.

Deterministic autoencoder

An autoencoder is a neural network that attempts to copy its input to its output via latent space $\mathbf{h}$. The network can be considered to consist of two parts: an encoder and a decoder. The encoder maps the input $\mathbf{x}$ to the latent space $\mathbf{h} = f(\mathbf{x})$. The decoder aims to reconstruct the inputs from the latent space $\mathbf{r} = g(\mathbf{h})$. The autoencoder is successful when $g(f(\mathbf{x})) = \mathbf{x}$. The value of a perfect autoencoder is however limited.

Typically, we aim to design the autoencoder in such a way that it is unable to perfectly reconstruct the inputs. That way, the autoencoder is forced to copy only input that resembles the training data, prioritizing useful properties of the data. Applications of autoencoders therefore include:

- Dimensionality reduction
- Feature learning
- Denoising

In this assignment you will need to create an autoencoder that operates on the idea that data concentrates around a low-dimensional manifold. The aim of autoencoders is to learn the structure of the manifold. More specifically, in this assignment you will use the MNIST database, a large public database of handwritten digits (0-9) that is commonly used for training various image processing systems.

Exercise 1

[4 pt] Create a convolutional autoencoder that loads batches of data from the MNIST dataset, for now only use the clean images. You can use PyTorch libraries and a sample code is already available for you to complete. Requirements for the network are as follows:

- You can only use Convolutional layers (Conv2D) or transposed convolutional layers, MaxPool2d, and Upsampling (using scale factor).
- For the encoder use Convolutional layers and for the Decoder transposed Convolutional layers
- After every (transposed) Convolutional layer, you must use rectified linear units (ReLU) as activation.
- Use a padding=1 for the (transposed) Convolutions
- After every (transposed) Convolutional (+activation) layer there must be a Pooling or Upsampling layer that decreases or increases, respectively, the dimension of the network by a factor 2
- (Transposed) Convolutional layers must all use 16 different filters with a kernal size of $(3, 3)$. Exceptions are the last layers of both the Encoder and Decoder (see below).
- Use two different functions, one for the Encoder and one for the Decoder
- The last steps in the Encoder should be designed such that the output of the Encoder has dimensions $[N \times 1 \times 2 \times 1]$, where N is the batch size. In other words, for each image, the latent space has dimensions $[1 \times 2 \times 1]$, the last 1 meaning that the last Convolutional layer in your Encoder should have only 1 filter and your last Pooling step should be designed such that it produces a $[2 \times 1]$ array/tensor.
- Train your autoencoder, using an MSE as loss.

Visualize the first 10 images of `example_clean` next to the output of this batch. What differences/resemblances you do notice between the input and output of the autoencoder? Also make a plot of the loss as a function of the number of iterations or epochs.

Exercise 2

(a) **[2 pt]**Create a Python script that visualizes the latent space that is provided by the Encoder for all the MNIST images in the test set. To this end, determine the output of the Encoder and make a scatter plot where you use the $[2 \times 1]$ vector of the latent space as values for the horizontal and vertical axes in a 2D scatter plot. Give each datapoint in the scatter plot a color that corresponds to the digit that is written in the original image (use `label`).

(b) **[0.5 pt]**Which digits provide clear clusters and which overlap? Can you think of an explanation for this?

(c) **[0.5 pt]**The datapoints, especially for some of the clusters, are clipped at weight of zero. Explain why this has happened.

Exercise 3

Although our network was not trained for classification, we can use the latent space to do some rudimentary classification of the handwritten digits in their correct class, classes being defined as the digits that are written in the images (i.e. `label`).

(a) **[2 pt]**Encode the test data `x_clean_test` and perform a 1-nearest neighbour search. You can use the sci-kit learn library for nearest neighbour implementations:
`from sklearn.neighbors import NearestNeighbors`
In the 1-nearest neighbour, for every datapoint in the latent space of the testset, you search the datapoint from the training set that has the smallest Euclidean distance in the latent space. You can consider this such that the two images are mapped to the same area in the latent space and hence have relatively high probability that their characteristics, hopefully the digits they represent, are similar. Therefore, assign the class to the test image that corresponds with the class `labels_train` of its nearest neighbour.

For the 1000 images in the testset, determine how many zeros, ones, twos, etc. were classified correctly and express these 10 accuracies as percentages of the total number of zeros, ones, twos, etc.

(b) **[1 pt]**If the classification based on the latent space would not work at all and produce random datapoint in the latent space, how many correct classifications would you expect? Which numbers are classified significantly better than the other, and (consequently) which number aren't? Can you explain this from the plot of the latent space you made in Exercise 2?

(c) **[0.5 pt]**If we would design our network differently, where our goal would not be to encode and decode the images but to correctly classify the images, what cost function would you use?

Exercise 4

As mentioned above, the autoencoder is not designed for classification, but still it manages to classify at least some of the digits relatively accurately. With a rather simple modification, it should be possible to convert your network into a network that is designed for classification purposes.

(a) **[2 pt]**Use your Encoder as starting point and replace the last Convolutional layer (and its associated Activation and Pooling layer) and replace these by a Fully-Connected, or its equivalent: Dense, layer. If necessary, you can use a Flatten and Activation layer as well. Assign a proper loss function and train and test your network on exactly the same training and test data.

What accuracies on the test set can you achieve now?

(b) **[1 pt]**Provide graphs of the training and test loss (perform intermittent evaluations of the test loss during training). Comment on the capacity of your network.

(c) **[1 pt]**Give some suggestions on improvements in case you were underfitting or in case you were overfitting.

Exercise 5

In this exercise you are going to use your previously trained autoencoder – in particular the Decoder stage – as a generative model.

(a) **[2 pt]**Define a 15 x 15 grid that equidistantly samples the latent space of Exercise 2 across both axes. In other words, if you visualize the latent space as a Cartesian coordinate system, sample between the points that describe the corners of your (cubic-shaped) latent space. Use 15 steps in both horizontal and vertical directions.

Use these samples from the latent space as inputs to your Decoder to create new images. Plot all images (`pyplot.imshow`) in one figure where you define the axes of the figure such that the grid coincides with the grid of the scatter plot in Exercise 2.

(hint: The directionality of plots and images are different. In a plot, the x-axis increases from left to right, and the y-axis increases from bottom to top. In an image, the x-axis also increases from left to right. The y-axis, however, is flipped: it increases from top to bottom. Compensate for this difference in your code for the best results)

(b) **[1 pt]**Which digits can you recognize? how do the images change from left to right and top to bottom? Can you explain why some images indeed look like digits, while others do not?

Exercise 6

Laslty, we want to see how well the trained autoencoder works for denoising purposes. The general idea behind that is as follows: because the autoencoder needs to pass information about the image though a bottleneck it will only encode semantically usefull information. During training it never needed to reconstruct noise, so during inference it should hopefully ignore it.

(a) **[2 pt]** Use the `x_noisy_example` as input to your autoencoder. Now create a figure conisting of 10 collumns and 3 rows. Each collumn should show one of the digits. Row 1 should show the noisy input, row 2 should show the output, and row 3 should show the corresponding clean image `x_clean_example`.

What do you observe? Can you explain what you see?

Variational autoencoder

Variational autoencoders (VAEs) are a special type of autoencoders with added constraints on the encoded representations that are being learned. More precisely, they are autoencoders that learn a latent variable model for the input data. So the autoencoder learns the parameters of a probability distribution that models your data. If you would sample points from this distribution, you could use the Decoder stage of the autoencoder to generate new samples of the input data. As such, you can consider variational autoencoders as so-called generative models.

Exercise 7

> It is recommended that you first follow the lecture on variational autoencoder, before you proceed with this Exercise.

(a) **[4 pt]**

- For this Exercise, the restriction on the Encoder and Decoder model do no longer apply. You can build new models, under the constraint that they must be Convolutional. Only the last layers of the Encoder and the first layer of the Decoder can be a Linear layer, if appropriate.

  To start the implementation, change the latent space of the Encoder model a bit. This was a single $[2 \times 1]$ vector per image in the batch. Change it such that your Encoder now has two $[2 \times 1]$ vector per image and refer to these vectors as the mean and standard deviation. In fact, typically we train a VAE to the log of the standard deviation or the log of the variance, so choosing a name accordingly is recommended.

- Create a function that takes the mean $\mathbf{x}_{mean}$ and (log) standard deviation $\mathbf{x}_{std}$ as inputs. Within the function, generate random numbers n, sampled from a normal distribution with the same dimensions as the (log) standard deviation. As output of the function, calculate new data points $\mathbf{x}_{out}$ that represent small deviations from the input to the function. These datapoints are defined as follows:

$$\mathbf{x}_{out} = \mathbf{x}_{mean} + n \cdot \exp\left(\mathbf{x}_{std}\right). \tag{.1}$$

  For all mathematical operations in the function, refrain from using numpy functions, but use only Pytorch. Add this function to your variational autoencoder. Hint: your encoder should now output 3 vectors: mean, (log) standard deviation, and sampled. Sampled should be used as input for the decoder, while the other two vectors are used in the loss function.

- You are now almost ready to train your VAE. However, to actually train a VAE we need to have a customized loss function that combines the loss you used for your deterministic autoencoder with a loss that forces ("regularizes") our learned distributions (i.e. the distribution that projects the data onto the latent space) to be as close as possible to a prior distribution (e.g. Gaussian). Create this custom loss function. This loss function consists of two terms, think about how you should balance these two.

- Train your VAE with this new loss function. Use at least 30 epochs.

Similar to Exercise 1, visualize the images in `example_clean` and also show what their respective output images are when mapped through the VAE. Moreover, also show the loss plot as a function of the number of iterations or epochs.

(b) **[1 pt]** Make the variational encoder predict the means of all the images in the test set and visualize this latent space similarly as you did in Exercise 2(a).

(c) **[1 pt]**Do you notice a difference with the visualization of Exercise 2? If not visually apparent, what is the main difference between the latent spaces of the deterministic and variational autoencoders?

(d) **[1 pt]**Repeat Exercise 5(a) for the VAE.

(e) **[1 pt]**What are the main differences with the result from Exercise 5(a)? Can you explain the differences between the generative model you used in Exercise 5 and the generative model based on the variational autoencoder?

Denoising with VAEs

As we alluded to at the start, VAEs can also be used for denoising. In this exercise we will denoise MNIST digits in two ways. First, we can use the information bottleneck of the VAE that, if well trained, should only encode information about the MNIST digits. Second, we can use the VAE as a prior to solve the denoising problem in a Bayesian manner. We can write the denoising problem as:

$$p(x_{estimate}|x_{noisy}) = \frac{p(x_{noisy}|x_{estimate})}{p(x_{noisy})} p(x_{estimate}), \tag{.2}$$

where $x_{estimate}$ are (estimated) MNIST digits, x_{noisy} is the (observed/measured) noisy MNIST digit. We now want to find the Maximum A-Posteriori (MAP) estimate for the estimated MNIST digit. This can be done using the following optimization procedure:

$$\arg\max_{x_{estimate}} \log(p(x_{noisy}|x_{estimate})) + \log(p(x_{estimate})). \tag{.3}$$

Note how we can ignore the evidence term $p(x_{noisy})$ and how we optimize in the log-domain (which changed the multiplication to an addition). We can now assume a Gaussian measurement error for $\log(p(x_{noisy}|x_{estimate}))$, and use the fact that the latent space of our VAE follows a Gaussian, to arrive at:

$$\arg\min_{z_{estimate}} ||x_{noisy} - g(z_{estimate})||_2^2 + \beta||z_{estimate}||_2^2. \tag{.4}$$

Note how this transforms the problem from finding the optimal image in pixel domain ($x_{estimate}$), to finding the optimal image in latent domain ($z_{estimate}$). If we solve the optimization problem outlined in equation (.4), we can find our final solution using: $x_{estimate} = g(z_{estimate})$.

Moreover, note how we have added a so-called Lagrange multiplier β. This hyperparameter is due to the fact that our prior $p(x_{estimate})$ and our likelihood function $p(x_{noisy}|x_{estimate})$, do not necessarily have the same variance. β is the parameter that controls for this difference. Another way to look at the parameter β is as a way to choose how to balance the importance of the likelihood vs the prior.

Exercise 8

To get the best results, change the size of the latent space of your VAE from 2 to 16 and retrain it.

(a) **[1 pt]** repeat exercise 6a using your new VAE.

(b) **[4 pt]** Now, implement equation (.4) using the decoder of your trained VAE. We can find the argmin through an optimization procedure. This time not on the parameters of a network, but on the values of the latent space. Using the noisy measurements of the 10 examples, try to create clean images using this algorithm.

Similar to 8a, show the noisy measurements you used as input, together with the reconstructions and the ground truth for the example images.

Moreover, also show a plot of the MAP loss over the iterations.

For implementation hints, take a look at `MAP_template.py`

(c) **[1 pt]** Out of the above two methods, which one works best. Can you give a reason why?