Deep Learning Project 3: Deep Generative Models

This documents describes the third and final assignment in IT3030, where you will implement deep generative models. The rules are as follows:

- The task should be solved individually.
- It should be solved in Python, implementing the functionalities described below.
- Your solution will be given between 0 and 10 points; the rules for scoring are listed in the last section.
- Demos for this assignment will be in Week 17: April 22-26.

1 Introduction

This assignment is about generative models - that is, models that are able to generate new examples of data that "look like" your training data. We will use two different models, that at least to some extent can be seen as generative models:

- Standard autoencoders
- Variational autoencoders (VAEs)

We will look at how each model works as a generator by considering some metrics defined below, and we will also look at how the generative models can work when we do anomaly detection. In this assignment you are allowed to use standard deep learning packages, and their probabilistic extension (Tensorflow with Tensorflow Probability or Pytorch with Pyro), but you should obviously not use already implemented (partial) solutions to the assignment, like an implementation of a VAE.

Your results should be presented during a demo session. To make your demo time effective, we ask you to make sure you have run your models **and saved the results** (both the plots you are asked to generate as well as the learned model weights) to files before the demo. Also take note of quantitative results you have obtained (like classification accuracy on generated data) **before** the demo, so you don't have to spend time on re-running heavy computations at demo time. During the demo session you will be asked to present the code and your design choices, show and discuss your results, and potentially use your saved model weights to recreate some of the results as required.

2 Supporting Code

2.1 Getting data

We will use different versions of the well known MNIST dataset for training the models. To this end, the supporting code in stacked_mnist[_tf].py can be useful. If you choose to use this

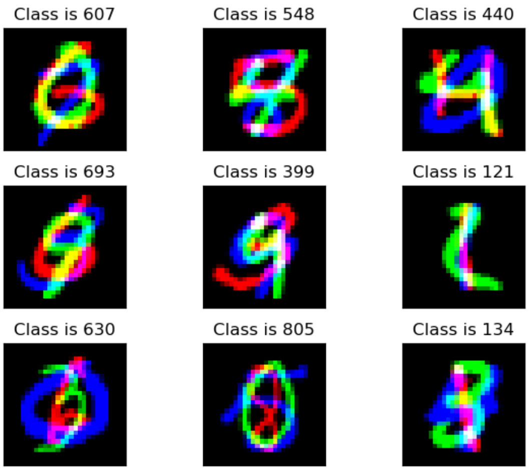

Figure 1: Examples of StackedMNIST images. Note that the Red channel decides the hundres, Green the tens, and Blue the ones, so the top left example with a red “6”, green “0” and blue “7” comes out as “607”.

code, you will instantiate the StackedMNISTData class with a DataMode. This gives you access to methods that provide data batches or the full dataset after preprocessing depending on the DataMode.

We make two versions of the supporting code available: one for Tensorflow and one for Pytorch. There are slight differences to how these are implemented, but the differences should be fairly self-explanatory by reading the code. The Pytorch version has not been tested extensively and may contain bugs.

The DataModes are of the form [MONO|COLOR] [BINARY|FLOAT] [COMPLETE|MISSING], and will be described below.

First and foremost, we will use two versions of MNIST: The classic grey scale images (digits are from 0 to 9), and a color version we refer to as *StackedMNIST*. An image from StackedMNIST is a color image where each color channel (Red, Green, Blue) is an MNIST image, see Figure 1. Since the image contains 3 images (one per channel), we will consider a color image from the dataet as an integer in the range [0, 999], and thus we have a dataset where each image can be classified into one of 1000 classes.

In the Tensorflow version, stacked_mnist_tf.py, you choose between the two by either using a DataMode that starts with MONO_XXX for monochrome images (that is, standard MNIST with ten classes) or COLOR_XXX if you want StackedMNIST (color images with 1000 classes); here the XXX part means that there are other things to decide upon, too, as described below. For instance will StackedMNISTData(DataMode.MONO_FLOAT_COMPLETE) give you access to the standard MNIST dataset.

In the Pytorch version, you select the mode by combining flags, so you get the standard MNIST dataset with StackedMNISTData(DataMode.MONO), and a color version where all the eights are removed from the training set with StackedMNISTData(DataMode.COLOR | DataMode. MISSING).

Next, you can choose to have the data binarized (pixel intensity values are 0 or 1) choosing a mode of the type XXX_BINARY_XXX, or with intensity values in the range $[0, 1]$ (with modes XXX_FLOAT_XXX). We recommend that you use the binarized version throughout, and only resort to the real-valued data if you are unable to get the binarized to work. Finally, we can choose between a dataset containing all digits in the training set, which we get using

XXX_COMPLETE, or a version where any number containing a digit "8" is taken out of the training data, using XXX_MISSING. You will use the latter when looking at anomaly detection. When doing anomaly detection in this project you will train your model on data where the digit "8" is not included, and an example from the test set where this digit is present is therefore something we hope the anomaly detector will react on.

Evaluation of a deep generative model

When evaluating the generative models we can obviously rely on human intuition: Plot the images, and check if they make sense. The eyeball test is important when you build your models. Later on, you'll want a way to automatically quantify how good a generative model is, and we shall use two approaches: Assessments of *quality* and *coverage*. The code for doing these quantifications are available through the VerificationNet defined in verification_net.py; the functionality is described below.

VerificationNet is a classifier that is capable of classifying data from a generative model, and simultaneously say something about how "certain" it is about each classification. As a proxy for manual inspection of generated images, it seems natural to use this classifier to check all generated examples: If the classifier selects a class with high confidence, then the generated image is of "good quality". One way to evaluate a generative model is therefore to do as follows:

1. Generate a large number of examples.
2. Classify each example, but instead of monitoring the classes, we will rather monitor the "confidence" in the most likely class.
3. The fraction of examples where the classifier's confidence is above some threshold indicates the generative model's quality.

Use VerificationNet.check_predictability(examples) to perform this test. If the images are reconstructions of original images from the training set, you can also check the accuracy of the reconstruction by checking if the generated examples are classified to the same class as the source original was: VerificationNet.check_predictability(examples, original_classes).

Tip 1: Eventually your system should work on RGB images (that is, using 3 color channels from StackedMNIST). However, initially it can be beneficial to try to solve the problem focusing on just a single color channel (that is, work with monochrome images from the original MNIST). The data generator has a setting for DataMode, which a.o. controls the number of channels. It may be very beneficial if you create the other parts of your code general enough for it to work in both monochrome and color, e.g., by letting each object get information about of the number of color channels as it is created. It is recommended to make sure you can solve the problem with one channel first, then move on to three channels when the simpler problem has been solved.

Tip 2: Spend some time thinking about the architecture for the models with 3 color channels. Can you utilize that each color channel in effect is an "independent part" of the full model, and that each of these parts in effect do the same as the one-channel model?

3 Make a classifier

The first step is to generate a classifier that can help us do the automatic validation of the generative models. Starting from the supporting code in verification_net.py, or your own setup, make a classifier that is trained to recognize MNIST and StackedMNIST numbers. It should be trained to have fairly good accuracy, say at least 98%, to ensure that your evaluations run smoothly. If you choose to build your own system, consider that each color can be assessed independently, that is, you only need a classifier for original MNIST images. If you simply use verification_net.py, just familiarize yourself with the code, choose a network architecture, and train until sufficient accuracy.

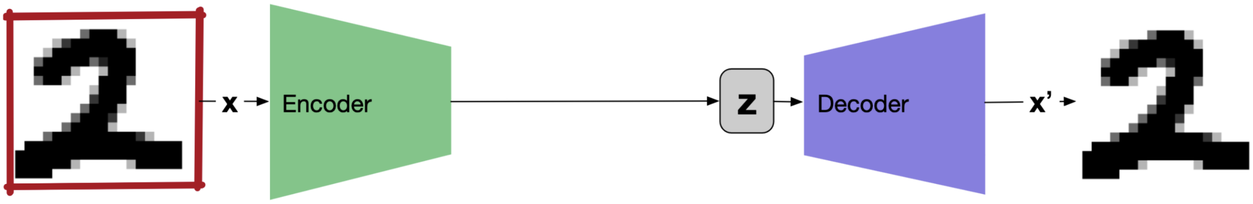

Figure 2: Autoencoders take some input data $\boldsymbol{x}$, here an MNIST image, and use the *encoder*-module to find a low-dimensional representation $\boldsymbol{z}$. The representation is chosen such that when passed through the *decoder* module, it can reconstruct the input to a certain level of quality.

4 The Autoencoder

We will first make a standard autoencoder. If you are not aware of what an autoencoder (AE) is, you are referred to the lecture on representation learning, where the AE and the steps for building it were described in detail. In general, the AE takes some input $\boldsymbol{x}$, and after encoding $\boldsymbol{x}$ as a low-dimensional representation $\boldsymbol{z}$, reconstructs the input to the best of its ability; we call the reconstruction $\boldsymbol{x}'$, see Figure 2.

Build an autoencoder that is able to work on both MNIST and StackedMNIST. It will be beneficial to use convolutions with stride > 1 in the encoder, and transposed convolutions for the decoder. If you use DataMode.XXX_BINARY_COMPLETE (where “XXX" must be changed to give either monochrome or color images), you will receive data objects where each value is either 0 or 1, and it is therefore advisable to use the binary cross entropy as reconstruction loss. If you for some reason need to use DataMode.XXX_FLOAT_COMPLETE, you must define an appropriate loss to go with that.

The AE model should be trained to a level where the reconstructions of images from the test-set are given the same class as the original images for at least 80% of the examples. Use tolerance .8 for one-channel images and .5 for the three-channel images. It is very beneficial during debugging and training of your model if you generate plots that show training data together with their reconstructions, so do that. These plots will also be needed during your demo session, so do save the plots to file.

4.1 AE as a generative model

Now we want to try the AE as a generative model. Sample random vectors for the encoding layer $\boldsymbol{z}$, and push the sampled values through the decoder part of the model. This gives you new $\boldsymbol{x}'$ values. It is not clearly defined by the AE model what distribution to sample from when generating $\boldsymbol{z}$, so you can choose this yourself (e.g., uniform on [0, 1] along each dimension of $\boldsymbol{Z}$, a Gaussian of some sort, or whatever you think is reasonable; a call like z = np.random.randn(num_samples, encoding_dim) should do it). The generated model will be in the same color mode as you used during training: If you train the model using StackedMNIST, this is what the model will try to generate, and if you used the one-channel monochrome images while training, you will also get monochrome images back.

Check quality and coverage as described above. Document your results by showing some of the generated images, similarly to Figure 3. What can you conclude regarding the autoencoder's abilities as a generative model?

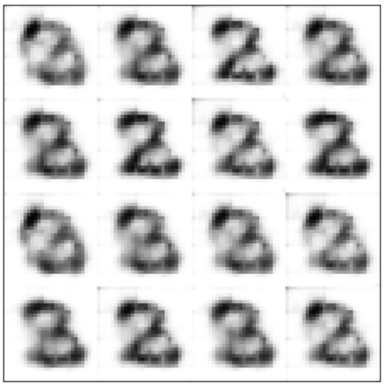

Figure 3: A typical result from the AE as a generative model. Notice that the tiling effect that is evident for some of the images is a consequence of using transposed convolutions with stride 2. The effect would probably disappear if the model had continued learning for some more epochs. The lack of variety and poor quality overall is in general not going away, though.

4.2 AE as an anomaly detector

One idea to detect anomalies in a dataset is to look at reconstruction error: It seems natural to expect the reconstruction loss to be higher for anomalous images than for "standard" images (again, refer to the lecture on learning representations if you do not understand this idea). Train the AE using data where one class is missing (using DataMode.XXX_MISSING when building the data source). Calculate the reconstruction loss when evaluating test data, and plot the most anomalous images. Did it work?

5 Variational Autoencoder - VAE

We now move on from the AE to its probabilistic extension, thereby having a first look at a *probabilistic AI system.* Probabilistic AI is discussed in two lectures, where we also get a fairly detailed look at the variational autoencoder that you will implement next.

A variational autoencoder is quite similar to a "standard" autoencoder, yet with a slight reinterpretation of the encoding, see Figure 4. As for an AE, the VAE has an encoder and a decoder part. However, for the VAE, the encoder determines a *distribution* over the encoding space instead of giving the encoding directly. The distribution is represented by the mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$, and with these determined, we can sample an encoding simply by first sampling a standard Gaussian variable, called $\boldsymbol{\epsilon}$, then $\boldsymbol{z} \leftarrow \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$, where $\odot$ is element-wise multiplication.

The VAE is potentially easier to understand as a probabilistic model, see Figure 4. The idea is that "Mother Nature" first selects a latent encoding $\boldsymbol{z}$, then this latent representation somehow "causes" the image $\boldsymbol{x}$. As you (may) remember about Bayesian networks, we need to define $p(\boldsymbol{z})$ and $p(\boldsymbol{x}|\boldsymbol{z})$ for this model to be fully specified. $p(\boldsymbol{z})$ is easy, we will just use a Gaussian distribution, but we will need to learn a representation of $p(\boldsymbol{x}|\boldsymbol{z})$ from data. We use a neural network to represent this, namely the decoder network of the VAE. With this interpretation of the decoder, the output related to a specific pixel gives the probability for that pixel being on.

Since we have a fully specified Bayesian network, we could in theory calculate the encoder part directly using Bayes' rule: $p(\boldsymbol{z}|\boldsymbol{x}) = p(\boldsymbol{x}|\boldsymbol{z}) \cdot p(\boldsymbol{z})/p(\boldsymbol{x})$, where $p(\boldsymbol{x}) = \int_z p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})\,d\boldsymbol{z}$. Unfortunately, we cannot calculate $p(\boldsymbol{x})$ efficiently in this situation, hence will use variational inference as an approximate solution for $p(\boldsymbol{z}|\boldsymbol{x})$ (and, since it is an approximation, it is denoted $q(\boldsymbol{z}|\boldsymbol{x})$ instead of $p(\cdot)$ in Figure 5); variational approximations and the VAE are discussed in detail in the ProbAI lectures. The encoder module implements the approximation: First we posit that $q(\boldsymbol{z}|\boldsymbol{x})$ is a Gaussian with mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$, then we let the encoder network generate $(\boldsymbol{\mu}, \boldsymbol{\sigma})$ for each $\boldsymbol{x}$ it is given as input.

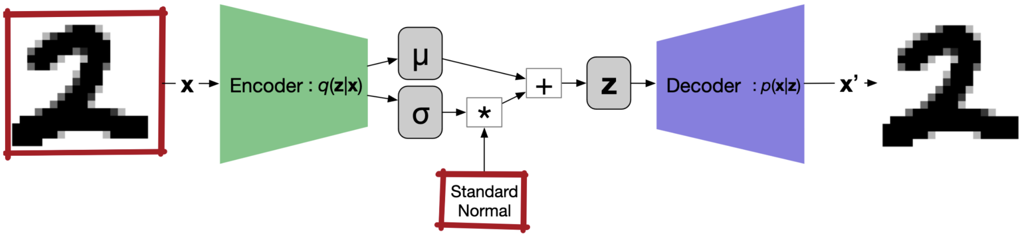

Figure 4: Variational autoencoders take (as AEs) some input data $\boldsymbol{x}$, and use the *encoder* module to find a low-dimensional representation. However, for the VAE, the encoder gives a *statistical distribution* for the representation $\boldsymbol{z}$. The representation is chosen such that when passed through the *decoder* module, it can reconstruct the input to a certain level of quality.

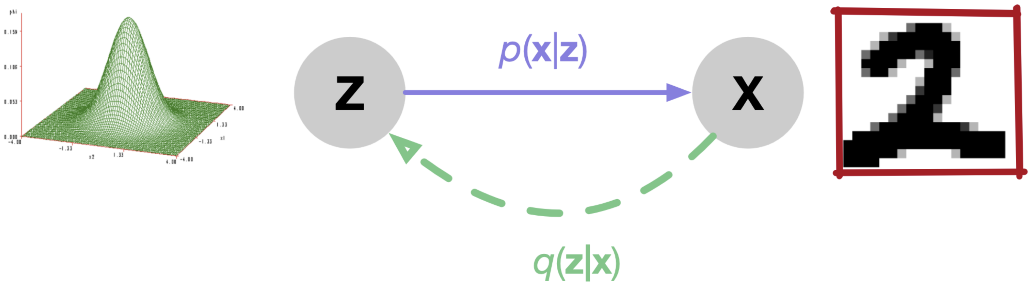

Figure 5: The encoder part of the VAE contains $q(\boldsymbol{z}|\boldsymbol{x})$, as an approximation of $p(\boldsymbol{z}|\boldsymbol{x})$.

5.1 VAE as a generative model

Do the same experiments as you did for the AE. Since the model explicitly models that $\boldsymbol{Z}$ follows the standard Gaussian distribution, you should sample from that distribution when feeding the decoder. You should not use the encoder to guide your sampling procedure and find some $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ that way - the aim is to let the model "dream", not reproduce some input.

Again, it is recommended to use convolutions and transposed convolutions as your main layers. The VAE model should be trained to a level where the reconstruction of images from the test set are given the same class as the original images for at least 80% of the examples. Use tolerance .8 for one-channel images and .5 for the three-channel images.

Compare the abilities of the ßAE and the VAE: Which has better generative properties when it comes to i) predictability, ii) coverage?

5.2 VAE as an anomaly detector

Before we describe this process, please note that **anomaly detection using VAE is different from that using AE**. The VAE defines an explicit probabilistic model over image space, and we can use that to improve the anomaly detection approach we pursued for the AE. As before, $\boldsymbol{X}$ represents an image, and it is therefore a tensor in three dimensions, with size given by (height-of-image, width-of-image, color-channels). Furthermore, we continue to use $\boldsymbol{Z}$ to represent a vector in the encoding space. Now, (check the lecture slides) the decoder is a probabilistic model for the process $\boldsymbol{Z} \rightsquigarrow \boldsymbol{X}$, i.e., represents the distribution $p(\boldsymbol{x}|\boldsymbol{z})$.

As discussed, $p(\boldsymbol{x})$ is difficult to calculate in general, but we can approximate it as follows:

$$p(\boldsymbol{x}) = \int_z p(\boldsymbol{x}|\boldsymbol{z}) \cdot p(\boldsymbol{z})\, d\boldsymbol{z} \approx \frac{1}{N} \sum_{i=1}^{N} p(\boldsymbol{x}|\boldsymbol{z}_{(i)}),$$

where $\boldsymbol{z}_{(1)}, \boldsymbol{z}_{(2)}, \ldots$ are N samples from $p(\boldsymbol{z})$, which was defined to be the standard Gaussian distribution. This actually buys us what we need: We can answer the question "How likely was this object $\boldsymbol{x}$, given the objects I have seen in my training data?" (In mathematical terms: "What is $p(\boldsymbol{x})$?") If the probability is low, the image is an anomaly. What we do here is to answer the question in two steps: First we ask "How likely is the image as a result for *one given encoding* $\boldsymbol{z}$?", then the second step is to average that one-encoding-result over many ($N \sim 10.0000$, say) possible encodings. Notice that $logp(\boldsymbol{x}|\boldsymbol{z}_{(i)})$ is readily available to us: Simply push a $\boldsymbol{z}_{(i)}$ through the decoder-part of the VAE, and calculate the binary cross-entropy loss between the output and the observed $\boldsymbol{x}$. Remember to use the exponential function to get back to $p(\boldsymbol{x}|\boldsymbol{z}_{(i)})$.

Implement this, and compare the VAE's ability to detect anomalies with what the AE could do. You will expect to see results as in Figure 6.

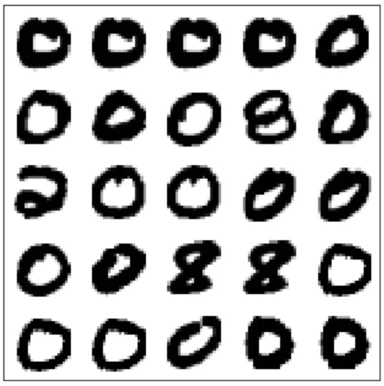

Figure 6: Anomalies found by VAE when the training data does not contain examples of the digit "8". In addition to some eights, the system also picks out atypical (thick-pen) version of some other digits. These results are generated after training for only 30 epochs, so your results will hopefully be better.

6 Earning points

The exercise can give you 10 points, and the table below lists which functionalities are required to get the different points.

Item	**Description**	**Points**
AE-BASIC	Implement the autoencoder, learn from standard MNIST data, and show reconstruction results.	1
AE-GEN	Show results for the AE-as-a-generator task on standard MNIST data. In addition to example images, the *quality* and *coverage* should also be reported.	1
AE-ANOM	Show results for the AE-as-an-anomaly-detector task on MNIST data. Show the top-k anomalous examples from the test set.	1
AE-STACK	Show the results for the AE-GEN and AE-ANOM tasks when learning from StackedMNIST data. Be prepared to discuss how you adapted the model structure when going from one to three color channels.	2
VAE-BASIC	Implement the variational autoencoder, learn from standard MNIST data, and show reconstruction results.	1
VAE-GEN	Show results for the VAE-as-a-generator task on MNIST data.	1
VAE-ANOM	Show results for the VAE-as-an-anomaly-detector task on MNIST data. **NOTE!** This is different from the AE-ANOM code. Simply doing the same as for the AE will give zero points.	1
VAE-STACK	Show the results for the VAE-GEN and VAE-ANOM tasks when learning from StackedMNIST data.	2
Total		10

WARNING: Failure to properly explain *any* portion of your code (or to convince the reviewer that you wrote the code) can result in the loss of points, depending upon the seriousness of the situation. This is an individual exercise in programming, not in downloading or copying. A zip file containing your commented code must be uploaded to Blackboard prior to your demonstration. You will not get explicit credit for the code, but it is crucial that we have the code online in the event that you decide to register a formal complaint about your grade (for the entire course).