IT3030 - Deep Learning

Assignment 3: Deep Generative Models

March 2020

This documents describes the third 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, implement the functionalities as described below.
- Your solution will be given between 0 and 20 points; the rules for scoring are listed in the last section.
- There is also a "bonus-question", worth up to five extra points. If you solve the bonus-question satisfactory, the extra points will count just as points gained elsewhere (hence, can e.g. help partly salvage a disappointing result from the exam, or from some of the other assignments).
- Deadline for **demonstration** is April 30th, during the announced lab-hours on that day. Upload the code to Blackboard before you demonstrate your code to an evaluator.
- Please note that grading depends on how you demonstrate your code to the evaluator, submission of code is just to lock your work for the event.
- Make sure code is commented properly and submitted to Blackboard before the demonstration starts.
- Lab hours/demonstration hours schedule will be published on Blackboard.
- Demonstrations will be online due to the ongoing Corona outbreak. The means for communication is given on Blackboard.

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 three different models, that at least to some extent can be seen as generative models:

- Standard auto-encoders
- Variational auto-encoders
- Generative Adversarial Networks

We will look at how each model works as a generator by considering some metrics defined below, and we will also look at how generative models can work when we do *anomaly detection*.

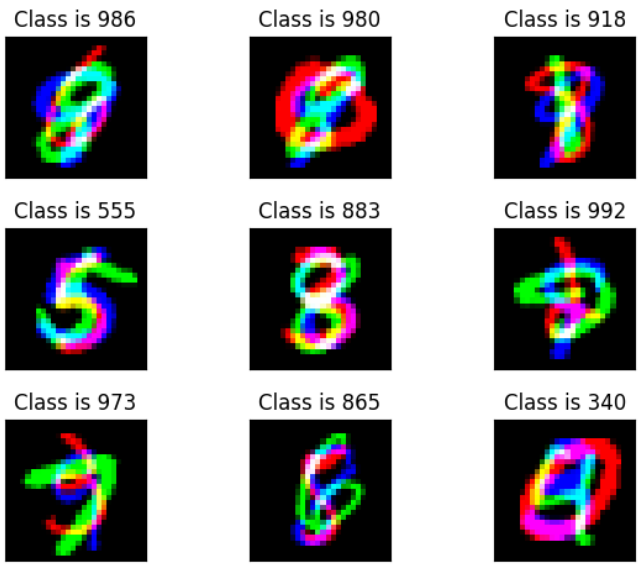

Figure 1: Some examples of stackedMNIST images. Note that the Red channel decides the ones, Green the tens, and Blue the hundreds, so the top left example with a red "6", green "8" and blue "9" comes out as "986".

Supporting code

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.py` can be useful. If you choose to use this 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. 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 known 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. 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.

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).

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.

We then train without seeing digit "8", thus an example from the test-set where this digit is used is an anomaly.

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 of *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 in the training set (as is the case for the auto-encoder and the variational auto-encoder), 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).

Furthermore, we will look for so-called *mode-collapse*, the situation when a generative model fails to generate examples from a specific "mode" in the data. In our case will define mode-collapse as the inability to generate all digits in the training-data (e.g., only generating digits "1" and "7", not the other classes in MNIST). We will look for mode collapse as follows:

1. Generate a large number of examples.
2. Discard all the images that are not of sufficient quality (i.e., that cannot be classified to a class with a predetermined "confidence").
3. Predict the classes for the remaining images, and find the number of *unique* classes that are generated
4. Class coverage is the percentage of possible classes that are actually generated

This functionality is available through VerificationNet.check_class_coverage(examples).

Keras tip: If you use Keras in your implementation you may need to define your own (non-standard) loss-function when building the VAE. The newest versions of Keras struggles with saving/loading weights for models with custom loss-functions. Hence, if you want to use Keras we recommend choosing Tensorflow v1. In particular Tensorflow v. 1.14.0 with Keras v. 2.2.4 seem to work; furthermore make sure to do "import keras" and **not** "from tensorflow import keras" when importing Keras.

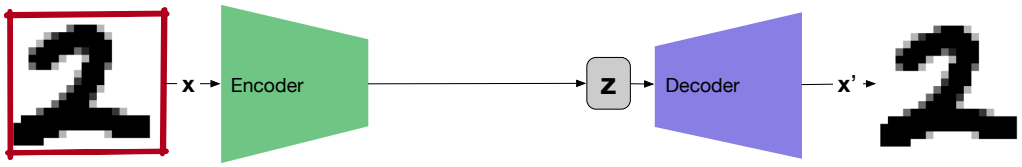

Figure 2: Auto-encoders take some input data $\boldsymbol{x}$, here an MNIST image, and uses the *encoder*-module to find a low-dimensional representation $\boldsymbol{z}$. The representation is chosen such that when passed through the *decoder* module, it can reconstruct the input to a certain level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

The auto-encoder

We will first make a standard auto-encoder. If you are not aware of what an auto-encoder (AE) is, you are referred to Assignment 2 and the slides from Lecture 7, 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 auto-encoder, 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.

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 of .8 in this experiment. It is very beneficial during debugging and training of your model if you generate plots that show training data together with their reconstructions.

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(no_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 auto-encoder's abilities as a generative model?

AE as an anomaly detector

One idea to detect anomalies in a data-set 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 Assignment 2 and Lecture 7 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).

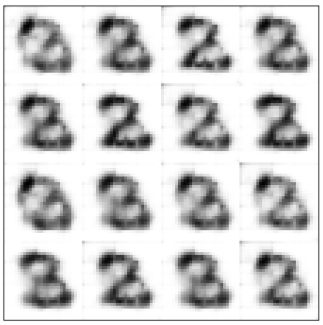

Figure 3: A typical result from the AE as a generative model. Notice that the tiling-effect that is evident for some of the images is a consequence of me using transposed convolutions with stride 2. The effect would probably disappear if I'd continued learning for some more epochs. The lack of variety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

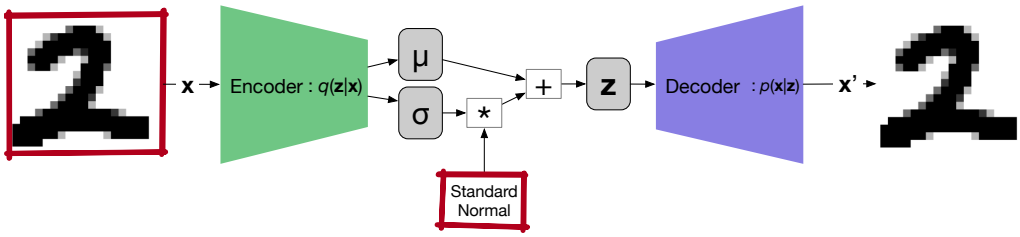

Figure 4: Variational auto-encoders take (as AE) some input data $\boldsymbol{x}$, and uses the *encoder*-module to find a low-dimensional representation. However, for the VAE, the encoder gives a *statistical distribution* for the representation $\boldsymbol{z}$. The representation is chosen such that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

Calculate the reconstruction loss when evaluating test-data, and plot the most anomalous images. Did it work?

Variational Auto-Encoder – VAE

A *variational* auto-encoder is quite similar to a "standard" auto-encoder, yet with a slight re-interpretation 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 variance $\boldsymbol{\sigma}$, and with these determined, we can sample an encoding simply by first sampling a standard Gaussian variable, called $\boldsymbol{\epsilon}$, then $\boldsymbol{z} \leftarrow \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$.

The VAE is potentially easier to understand as a probabilistic model, see Figure 5. 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 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

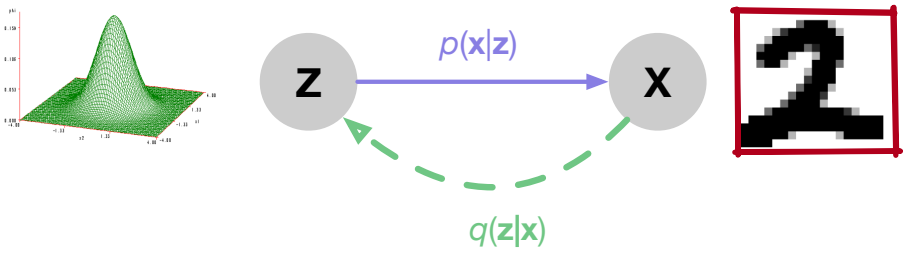

Figure 5: Variational auto-encoders take (as AE) some input data $\boldsymbol{x}$, and uses the *encoder*-module to find a low-dimensional representation. However, for the VAE, the encoder gives a *statistical distribution* for the representation $\boldsymbol{z}$. The representation is chosen such that when passed through the *decoder* module, it can reconstruct the input to a certain level of quality.

directly using Bayes' rule: $p(\boldsymbol{z}|\boldsymbol{x}) = p(\boldsymbol{x}|\boldsymbol{z}) \cdot p(\boldsymbol{z})/p(\boldsymbol{x})$, where $p(\boldsymbol{x}) = \int_{\boldsymbol{z}} p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})\, d\boldsymbol{z}$. 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 will be discussed in detail during the remaining 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 generate $(\boldsymbol{\mu}, \boldsymbol{\sigma})$ for each $\boldsymbol{x}$ it is given as input.

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. Compare the learned models' abilities. 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 of .8 in this experiment.

VAE as an anomaly detector

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

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

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

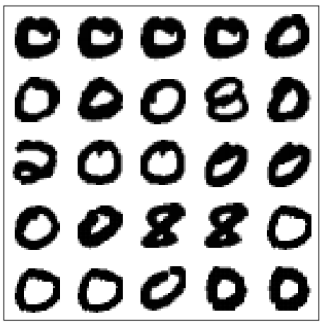

Figure 6: Anomalies found by VAE when the training data does not contain examples of the digit "8". In addition to some eights, the system also picks out atypical (thick-pen) version of some other digits. These results are generated after training for only 30 epochs, so your results will hopefully be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

question in two steps: First we ask "How likely is the image as a result of a given encoding $\boldsymbol{z}$?", then average that over many ($N \sim 100.0000$, say) possible encodings. Notice that $p(\boldsymbol{x}|\boldsymbol{z}_{(i)})$ is readily available to us: Simply push a $\boldsymbol{z}_{(i)}$ through the decoder-part of the VAE, and calculate the cross-entropy loss between the output and the observed $\boldsymbol{x}$.

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.

Create a Deep Convolutional GAN – DCGAN

Generative Adversarial Networks (GANs) have created a lot of buzz lately, with an impressive ability to generate objects with extremely high quality. In this assignment we will use a simple DCGAN (as will be discussed in the lectures). The main idea of a GAN is that we have two models duelling: A *generator* and a *discriminator*. The generator takes in random noise, and produces data (like images that may be regarded as similar to those in the MNIST dataset). The discriminator takes in an object, not knowing in advance whether it is a real data point (something from the training data) or something the generator has produced. The discriminator's goal is to be good at separating fake data from real data, whereas the generator attempts to fool the discriminator. During training, the two models co-evolve, and after training, the generator takes centre stage, as it has learned to transform random noise into realistic data objects (like, e.g., MNIST-like images).

In this assignment you shall implement DCGAN (see `https://arxiv.org/pdf/1511.06434.pdf`), for instance using Keras. DCGAN is like a "classic" GAN, but as you'd guess from the name, DCGAN employs convolutions. There are some tricks to note: Use convolutions with strides > 1 instead of max-pooling, and transposed convolutions with strides > 1 instead of upsampling. Batch normalization is recommended almost everywhere, and pay attention to which activation functions to use where; for instance the paper recommends to use *leaky* ReLU in the discriminator.

You'll need to show examples of generated images, and check image quality and coverage as above.

There is no obvious way to use GANs for anomaly detection, hence we don't do that in this assignment.

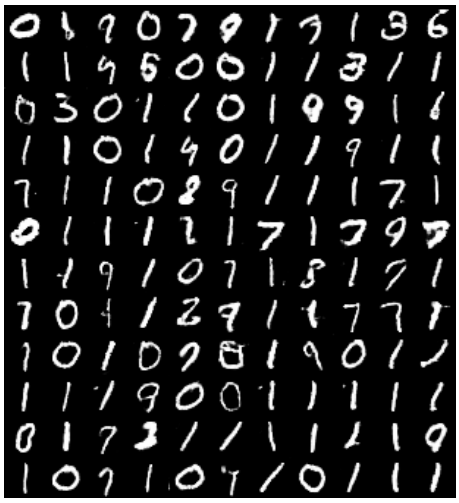

Figure 7: MNIST-like images generated by DCGAN after 100 epochs.

Bonus challenge

As you may have observed, mode collapse can be a serious issue for GANs. Results like the generated dataset in Figure 7 are typical; notice that each digit is clearly represented, most are of decent quality, but there is an over-reliance on specific digits (in this case the digit "1").

In practical terms, if not being told otherwise, the generator is constantly looking for the discriminator's most accessible weaknesses, instead of trying to faithfully represent the full splendor of the training data (e.g., if it can fool the discriminator when making a representation of the number 1, there is no need to waste efforts on making impressions of other numbers). In the extra-challenge you should investigate techniques for GANs to avoid mode-collapse. This problem is close to bleeding edge Deep Learning research, hence the bonus-challenge is deliberately under-specified. There are no strict requirements on how to solve the challenge, you can either read paper(s) on the topic and write a two-page summary that you present to the assistant, or show improvements through experiment. You are allowed to download and use any software package you find online (as long as it was not prepared by other students in this course), and adapt it to your needs.

Note that this bonus challenge will only release points after a certain point-score has been earned from other parts (see below).

Earning points

The exercise can give you up to 20 points (plus bonus), and the table below lists which functionalities are required to get the different points.

Item	Description	Points
AE-BASIC	Implement the auto-encoder, learn from standard MNIST data, and show reconstruction results.	2
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.	2

Continued on next page

Item	Description	Points
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.	1
VAE-BASIC	Implement the variational auto-encoder, learn from standard MNIST data, and show reconstruction results.	4
VAE-GEN	Show results for the VAE-as-a-generator task on MNIST data.	2
VAE-ANOM	Show results for the VAE-as-an-anomaly-detector task on MNIST data.	2
VAE-STACK	Show the results for the VAE-GEN and VAE-ANOM tasks when learning from stackedMNIST data.	1
GAN-BASIC	Implement the DCGAN, learn with MNIST data, and generate new images. Document the results through images, as well as quality and coverage numbers.	4
GAN-STACK	Show the same results as in GAN-BASIC, but now learning from stackedMNIST data	1
Total		20

Note! The bonus-question will give up to 5 additional points, based on your investigations. You will only get credits for the bonus question if all the other questions have been at least partly successful, and you have obtained at least 16 of the 20 points from the "standard" part of the assignment (e.g., you will not get bonus points if you completely failed to implement the GAN).

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 nor 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).