Final Project

GRA41522 OOP with Python

Autumn 2025

Instructions

Read carefully: Create a single pdf file with your answers, which are limited to 8 content pages. Make sure to provide the link to your **anonimyzed** GitHub repository where you pushed all relevant code in this project. Your report must contain: i) a front page with your ID as in WISEflow, ii) questions with numbers as header, and iii) page numbers. Your answers should be brief and concrete, **and you should always support all your claims. Answers that are not well supported may not be counted as correct.**

Grading Criteria: We will evaluate how well concepts like inheritance, overriding, polymorphism, etc. are implemented and theoretically justified in the report. **The lack of theoretical justifications may significantly affect the score**. An example of poor implementation is the inadequate use of inheritance, overriding, polymorphism, etc., which creates redundant code. The code in the GitHub repository should reproduce the results in this report in a fully automated way, failing to do that will affect the score. That said, your grade is based on your report and your code. If you have difficulty with the models involved in the project, you can ask me for help. However, depending on how much assistance you require, I may deduct some points from your total score.

Honor Code: By accepting this project, I confirm that I will not seek or provide assistance from any **person** or **AI tool**, as this is considered cheating. **Any suspected cheating** will be reported immediately to the exam administration and students will be called for an oral consultation as an additional verification before receiving a final grade. Good luck!

Variational Autoencoders

Variational Autoencoders (VAEs) are powerful models in generative artificial intelligence as they combine deep learning and probabilistic modeling, which enables generative modeling and representation learning. Unlike traditional autoencoders, VAEs learn a latent space $\boldsymbol{z} \in \mathbb{R}^d$ by assuming a variational posterior distribution $q(\boldsymbol{z}|\boldsymbol{x})$, where $\boldsymbol{x}$ is the input data, and a prior distribution $p(\boldsymbol{z})$. Therefore, VAEs offer a probabilistic approach to learning meaningful and disentangled representations and are essential for problems such as image synthesis, anomaly detection, and representation learning, among other learning tasks. VAEs' generative capabilities arise from simultaneously training the distribution $p(\boldsymbol{x}|\boldsymbol{z})$ and assuming an appropriate distribution for the given input data $\boldsymbol{x}$.

VAEs are relatively simple to train, and their objective function stems from maximum likelihood estimation (MLE). For most VAEs, however, a direct MLE approach is intractable, so we need to approximate the log-likelihood by a lower bound. This lower bound is given by

$$\mathcal{L} = \mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x})}[\log p(\boldsymbol{x}|\boldsymbol{z})] - KL[q(\boldsymbol{z}|\boldsymbol{x})|p(\boldsymbol{z})], \tag{1}$$

where KL denotes the Kullback-Leibler divergence, and becomes the objective function that we maximize in practice.

The common choice for the distribution $q(\boldsymbol{z}|\boldsymbol{x})$ is multivariate Gaussian with diagonal covariance matrix, and for the prior $p(\boldsymbol{z})$ is an isotropic Gaussian distribution $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$. The parameters of the distribution $q(\boldsymbol{z}|\boldsymbol{x})$ are learned by deep neural networks, i.e.

$$q(\boldsymbol{z}|\boldsymbol{x}) \sim \mathcal{N}(\boldsymbol{z}|\boldsymbol{x}; \boldsymbol{\mu} = f_{\boldsymbol{\phi}}(\boldsymbol{x}), \boldsymbol{\sigma}^2 = f_{\boldsymbol{\phi}}(\boldsymbol{x})), \tag{2}$$

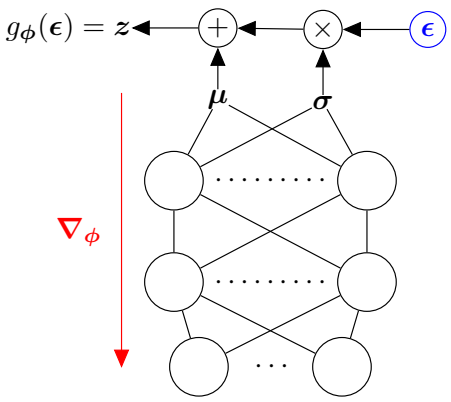

Figure 1: Sampling $\boldsymbol{z} \sim q(\boldsymbol{z}|\boldsymbol{x})$ is achieved by drawing $\boldsymbol{\epsilon} \sim \mathcal{N}(0, 1)$, multiplying it by $\boldsymbol{\sigma}$ and adding the product to $\boldsymbol{\mu}$.

where $f_{\boldsymbol{\phi}}(\cdot)$ is a deep neural network with trainable parameters $\boldsymbol{\phi}$. Given a trained deep neural network $f_{\boldsymbol{\phi}}(\cdot)$ parameterizing $q(\boldsymbol{z}|\boldsymbol{x})$, sampling $\boldsymbol{z}$ is done by the location-scale sampling method $\boldsymbol{z} = \mu + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$, where $\epsilon \sim \mathcal{N}(0, 1)$ and $\odot$ denotes element-wise multiplication, see Figure 1.

Note that the distribution $q(\boldsymbol{z}|\boldsymbol{x})$ takes the input data $\boldsymbol{x}$ and learns a latent representation $\boldsymbol{z}$, which is further used as input of the generative model $p(\boldsymbol{x}|\boldsymbol{z})$ to sample new data $\boldsymbol{x}$ using the location-scale approach, see Figure 2. Therefore, we call the former distribution *probabilistic encoder* and the latter distribution *probabilistic decoder*. It is worth noting at this point that the encoder minimizes the KL divergence in Equation 1, whereas the decoder maximizes the log-density, which is evaluated using the Monte Carlo approximation $\frac{1}{L} \sum_{l=1}^{L} \log p(\boldsymbol{x}|\boldsymbol{z}_l)$. The density $p(\boldsymbol{x}|\boldsymbol{z})$ is chosen depending on the type of data, e.g. pixels in colored images are assumed to have a multivariate Gaussian distribution.

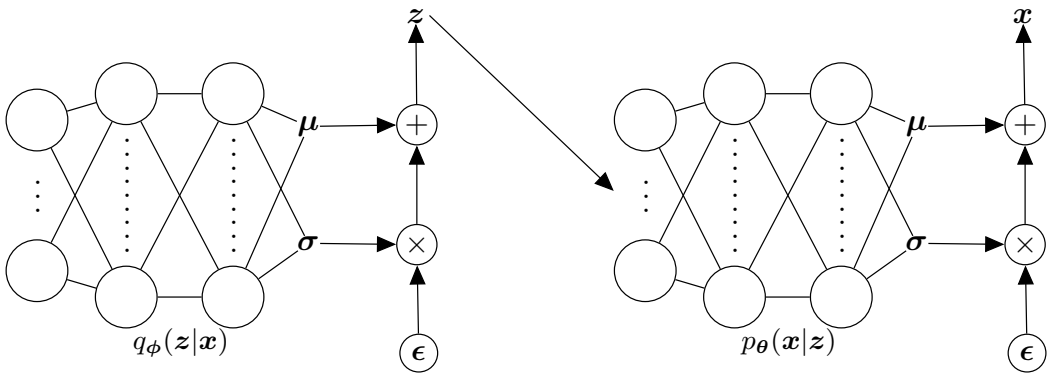

Figure 2: Architecture of a VAE.

1 Implementing a VAE (70 pts.)

Your task is to implement a VAE using object oriented programming (OOP). To that end, you are asked to construct the following classes:

a) BiCoder: this class should contain the common behavior in encoders and decoders.

b) Encoder: this class should contain the specific behavior in encoders.

c) Decoder: this class should contain the specific behavior in decoders.

d) VAE: this class should contain the behavior in VAEs.

e) OOP core concepts: When developing the classes in the previous exercises, make sure you use the following concepts/methods:

- Inheritance
- Abstract methods
- Polymorphism
- Overriding
- Decorators

Explain each concept, stating in which class(es) or method(s) it is used, and justifying your approach. Note, the actual neural network architectures, as well as some useful `TensorFlow` commands/functions, are in the folder `Project2025` in the course `GitHub` repository.

f) Testing your code: Write a program called `train_vae.py` that trains your VAE using the datasets `mnist_color.pkl` and `mnist_bw.npy`, which you will find in the folder `Project2025` in the `GitHub` repository of the course. Your program must:

1. Train the VAE model
2. Generate the latent space $\boldsymbol{z}$ and visualize it using the libraries `sklearn.manifold.TSNE` and `matplotlib.pyplot.scatter`, which should be downloaded to the conda virtual environment.
3. Generate new images sampling $\boldsymbol{z}$ from the prior $p(\boldsymbol{z})$.
4. Generate new images but this time sampling $\boldsymbol{z}$ from the posterior distribution $q(\boldsymbol{z}|x)$.

Make sure to use the library `argparse` so the code can be run in a flexible way from the command line, e.g.

```
python train_vae.py --dset mnist_bw --epochs 20 --visualize_latent
python train_vae.py --dset mnist_bw --epochs 20 --generate_from_prior
python train_vae.py --dset mnist_bw --epochs 20
                                            --generate_from_posterior
```

When coding `train_vae.py`, make sure you use as many of the concepts we have learned in the course as possible and explain/justify your approach.

2 DataLoader Class (30 pts.)

Although it is easy to load the datasets `mnist_color` and `mnist_bw` into the in-build `Tensorflow` data loader `tf.data.Dataset.from_tensor_slices()`, it is better to have your own class(es) to handle the data. Your task is to design a class, or classes, to accomplish the following tasks:

1. Download the dataset from the links in folder `Project2025` that is in the `GitHub` repository of the course. Note, `wget` is the safest in this case.

2. Unzip the data `mnist_color`, which doesn't need further preprocessing and it is a dictionary with 5 different versions of MNIST digits in color. You can choose any of the version using the following keys: 'm0', 'm1', 'm2', 'm3', and 'm4'.
3. Apply the following transformation to `mnist_bw`:
   - Scaling: make the image be in the interval 0 and 1 by dividing the data by 255.
   - Vectorization: each image is a vector with `28*28=784` dimensions. Therefore, the shape of the data becomes `(60000,784)`.
4. Load and return the processed data in a `tf.data.Dataset.from_tensor_slices()` data loader that can be used in the method `call` to train your VAE.

Implementation Tips

Below you find some help for some of the methods that you need to implement in your the neural network classes.

`def __init__`: You must define all variables you need for the rest of the methods. The variables that you define in the constructor can be available in all other methods without needing to be passed as arguments.

`def call`: This is a special method in `TensorFlow`, which is used to specify all steps to arrive to the objective function to be optimized, e.g. Equation 1. Commonly, the only input parameter is the input data for your model.

`def train`: This method updates the network's trainable variables and commonly takes `x` and `optimizer` as input parameters, where `optimizer` can be defined as `tf.keras.optimizers.Adam(learning_rate=1e-3)`. See the actual code for this method in `utils.py`.

`Inheritance:` Depending on the architecture of your application, BiCoder, Encoder, and/or Decoder classes must inherit from `layers.Layer`, where `layers` is defined as `from tensorflow.keras import layers`. Similarly, your `VAE` class must inherit from `tf.keras.Model` where `tf` is defined as `import tensorflow as tf`. Failing to follow these steps will cause the method `train` to crash, as `self.trainable_variables` will not be traced.

`Neural Networks:` You can think of the neural networks in the file `neural_networks.py`, e.g. `encoder_mlp`, as an instance of the `Sequential` class. This object can be called by passing data `x` (or a minibatch) as an argument, i.e., `out = encoder_mlp(x)`, which are the outputs depicted in Figure 1. $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. Since $\boldsymbol{\sigma}$ is strictly positive, in practice we assume that the output of the neural network is $\log(\boldsymbol{\sigma}^2)$. Therefore, $\boldsymbol{\sigma} = \exp(0.5 * \log(\boldsymbol{\sigma}^2)$, as you can see in `neural_networks.py`.

Debugging Tips

If you are struggling to train your VAE, start by debugging the BiCoder, Encoder, and Decoder classes. If properly coded, you should be able to create objects and call some of their methods before training the VAE. Although it is not a good practice, you can try to access private variables of the objects to see if they are as they are supposed to be, e.g. printing their shapes. Remember the inheritance tips.

After you are sure that your BiCoder, Encoder, Decoder, and VAE classes are correct, start debugging the data loaders class. Follow the same approach, that is, define an object and check that the methods are working as expected. At this point, you are sure that the VAE and data loader classes are correct. Then you can focus on the last step, which is training your VAEs. You can think of this last step as a *tester* file to verify that you did everything correctly. The `train_vae.py` file is straightforward and does not require

many lines of code. You need to i) load the data set, ii) initialize your VAE model, iii) set the optimizer, iv) invoke the method `train` using a mini batch from your `DataLoader`. See the pseudo-code below.

```
my_data_loder = DataLoader(dset=args.dset)
model = VAE() # use default values, or pass arguments with argpase
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
tr_data = my_data_loder.get_training_data
for e in range(args.epochs):
  for i, tr_batch in enumerate(tr_data):
    loss = model.train(tr_batch, optimizer)
```