1 Introduction

In this assignment, it is expected that you explore autoencoders in reconstruction and denoising tasks, and exploit an object detector network with real world data. This assignment is split in two parts: image reconstruction and object detection. This assignment is part of your continuous evaluation. You should provide a report and notebooks (.ipynb), which include your development and analysis for the requested tasks.

2 Dataset

In this assignment you will be using the MNIST dataset (used in the first CNN code) and a subset of the KITTI[1] object dataset. The KITTI dataset was recorded in outdoor environments using an ego-vehicle equipped with different sensors, including an RGB camera. For the purpose of this work, a KITTI subset containing 500 raw RGB images (the images are not object-centric images) as well as their labels for the 3 object classes (Car, Person, and Cyclist), was prepared, and it is available to download at https://drive.google.com/file/d/1xMKIf6igwCYWMFTjsovlt04rJwgWUhbj/view. A raw RGB image may contain more than one object, where each row in the label file represents an object (bounding box) with the following structure:

```
object_class_id normalized_center_x normalized_center_y normalized_width normalized_height
```

It is important to note that an image can contain no objects.

3 Part I: Reconstruction and denoising using Autoencoders

3.1 Background Materials

In this first part of the assignment, you will define a suitable network composed by encoder and decoder architectures to process MNIST images. You will implement an Autoencoder (AE), a Variational Autoencoder (VAE) and a Denoising Autoencoder (DAE).

3.1.1 Autoencoders

Implementing AEs is a simple process in PyTorch. Similarly to how a network is built, you must first define the encoder layer and decoder layer and later call them in the "**forward(self,input)**" function. You must also define a loss function that measures how similar (or how distant) the decoding and the input are. Commonly, Mean Squared Error (MSE) and Binary Cross Entropy (BCE) are used as loss functions for reconstruction problems. The training code from the previous assignment can be slightly modified to suit a reconstruction task, taking into account that the loss now considers the batch images and the batch decodings. It is important to note that the last activation function in the decoder layer plays an important role in the reconstruction task, as it should map the pixel values correctly. Although it is possible to create encoder/decoder layers using linear/fully connected layers, since we are working with image data, convolutional encoder/decoder layers should be used. You are not required to make the encoder and decoder a complete mirror of each other (e.g., MaxPool2D is not fully invertible), however, for most cases, "inverse" layers are required (Conv2d $\rightarrow$ ConvTranspose2d).

3.1.2 Variational Autoencoders

VAE is a type of generative model that combines elements of AEs and probabilistic modeling, meaning that from an initial AE network, modifications can be introduced to build a VAE network. The encoder layer of the AE, outputs a deterministic representation of the feature map, also known as bottleneck. To implement a VAE, the first step will be to modify the encoder's output layer to output the mean (μ) and

[1] http://www.cvlibs.net/datasets/kitti/

the log-variance ($\log \sigma^2$) of a probability distribution over the latent space. The problem now becomes how to sample from the distribution while maintaining the network traversable during back propagation, as the efficient calculation of gradients during the backpropagation process is essential for training neural networks. This is known as the reparameterization trick and consists in adding a sampling layer that generates random samples taken from a normal distribution and scales it by the standard deviation (represented by the log-variance). The latent variable z is modeled as a Gaussian distribution with mean μ and standard deviation σ, and ϵ is a sample from a standard normal distribution ($\epsilon \sim \mathcal{N}(0, 1)$). Since the VAE learns the log-variance (as it improves numerical stability) the reparameterization trick is expressed as:

$$z = \mu + e^{0.5(\log \sigma^2)} \cdot \epsilon \tag{1}$$

The input of the VAE's decoder layer becomes the sampled latent vector. Finally, the total loss for the VAE is composed of two parts, the reconstruction loss and the KL Divergence loss. The reconstruction loss is the loss used in the AE. The KL Divergence loss is introduced to evaluate the distance between the learned distribution and a chosen prior (commonly a standard normal distribution). The KL Divergence loss is expressed as:

$$D_{KL} = -\frac{1}{2}\sum_{i=1}^{N}(1 - \mu_i^2 + \log \sigma_i^2 - e^{\log \sigma_i^2}) \tag{2}$$

where N is the dimensionality of the latent space.

3.1.3 Denoising Autoencoders

The main goal of a DAE is to learn how to denoise the input data, which can improve its ability to extract meaningful features. For that reason, the only step required is the generation of noisy inputs:

```
def add_noise(data, noise_factor=0.5):
    noisy_data = data + noise_factor * torch.randn_like(data)
    return torch.clamp(noisy_data, 0., 1.)
```

3.1.4 Evaluation Metrics

Since Autoencoders are data-driven, the evaluation of your models will be in terms of reconstruction errors and the qualitative representation of some reconstruction pairs.

3.2 Tasks

- Implement an AE network (the network of the first assignment can be a good starting point); You can use a fully-connected-based encoder-decoder, however a convolution-based encoder-decoder will be valued.
- Modify the encoder, add the reparametrization trick and add the VAE loss to the aforementioned network;
- Implement the DAE's pipeline;
- Implement the aforementioned evaluation metrics to evaluate your network;
- Gather the metrics and loss curves for all training conditions.
- Use t-SNE (t-distributed Stochastic Neighbor Embedding) to visualize the bottleneck/latent space of the AE, VAE and DAE. Example:

```
latent_space = []
labels = []
with torch.no_grad():
    for batch_idx, data in enumerate(training_loader):
        ae.eval()
        mu= ae.encode(data[0].to(device))
        latent_space.append(mu)
        labels.append(data[1])
```

```
latent_space = torch.cat(latent_space, dim=0).cpu().numpy()
labels = torch.cat(labels, dim=0).cpu().numpy()

from sklearn.manifold import TSNE
tsne = TSNE(n_components=2, random_state=42)
latent_tsne = tsne.fit_transform(latent_space)

import matplotlib.pyplot as plt
plt.scatter(latent_tsne[:, 0], latent_tsne[:, 1], c=labels, cmap='viridis')
plt.colorbar()
plt.title("t-SNE Visualization of Latent Space")
plt.show()
```

;

- Apply transfer learning using the encoder layer of the AE, VAE and DAE. Add a classification layer to all the encoders (individual), freeze the encoder layer and only train the classification layer. Compare the classification results obtained, and how well the learned representations generalize.

4 Part II: Object Detection

4.1 Background Materials

In this second part of the assignment, you will use the object detector YOLOv5[2] network. YOLO is one of the most successful real-time CNN-based object detectors developed to date.

4.1.1 Evaluation Metrics

Evaluating an object detection network is not the same as evaluating an object classification one. In an object detection network, the prediction results are bounding boxes within an image representing the detected objects, instead of a category label associated with an image. These predicted bounding boxes need to be evaluated by the Intersection over Union (IoU) metric, which evaluates the precision of each predicted bounding box with the ground-truth. IoU is represented by (3). The individual bounding box evaluation is the base of the object detection method evaluation metrics, designed by mean-Average Precision (mAP), which is based on Precision and Recall evaluation metrics as shown in (4), (5), (6), and (7).

$$IoU = \frac{area\ of\ overlap}{area\ of\ union} \tag{3}$$

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

$$AP = \frac{\sum_{r \in Recall([0,1])} Precision(r)}{|Recall([0,1])|} \tag{6}$$

$$mAP = \frac{\sum_{c \in C} APc}{|C|} \tag{7}$$

where:

- True Positive (TP): correct class prediction and usually IoU > 0.5;
- False Positive (FP): wrong class or IoU < 0.5;
- False Negative (FN): missed object detection.

[2] https://github.com/ultralytics/yolov5

4.1.2 Environment Setup

The following guideline shows you the steps that you must perform to run YOLOv5 on Colab:
- First, download the YOLOv5 implementation as well as the KITTI subset prepared for this assignment (https://drive.google.com/file/d/1xMKIf6igwCYWMFTjsovlt04rJwgWUhbj/view);
- Unzip the file and copy it to you Google Drive;
- Mount your Google Drive account on the Colab notebook;

```
from google.colab import drive
drive.mount('/content/gdrive')
```

- Change the running directory of the notebook to the 'YOLO' folder;

```
%cd gdrive/MyDrive/YOLO/ # Example of the YOLO folder path
!git clone https://github.com/ultralytics/yolov5  # clone repo
%cd yolov5
%pip install -qr requirements.txt # install dependencies
```

- Run the 'train.py' to train the YOLOv5 network or run the 'detect.py' file to observe the trained YOLOv5 model.

```
(pretrained) !python train.py --img 416 --batch 16 --epochs 50 --data DATASETFILE --weights
    WEIGHTSFILE --cache
(scratch) !python train.py --img 416 --batch 16 --epochs 50 --data DATASETFILE --weights '' --cfg
    NETWORK.yaml --cache
```

4.2 Tasks

- Prepare the data configuration (yaml file to set DATASETFILE) files required to run the YOLOv5 network on the provided KITTI subset. For that, you should use an 80/20% split for training/testing, respectively; DATASETFILE example (data/APA2023.yaml):

```
train: YOLO/KITTI_dataset/Dataset/train    # folder contains images and labels subfolders
val: YOLO/KITTI_dataset/Dataset/valid      # folder contains images and labels subfolders
nc: 1
names: [Vehicles]
```

- Select one of the architectures available on the YOLOv5's GitHub webpage (e.g., YOLOv5m or YOLOv5s) and train the YOLOv5 network using only the "object class 0 (cars)". Train the network from scratch during 50 epochs (it will take a while);
- Train the same YOLOv5 architecture using as a starting point the pretrained checkpoints provided on the YOLOv5's GitHub webpage (use also 50 epochs).
- Gather the metrics and loss curves for both training conditions.

5 Deliverables

Each group (two students) must deliver the following material:

- A detailed report (**cannot exceed 10 pages**) with:
  - **Part I**:
    - A theoretical comparison between each AE, VAE and DAE network. For that, present a diagram/table of each network architecture developed;
    - Report all hyperparameters used;
    - Plots of the attained loss curves, sets of images that represent the reconstruction capabilities of each network (input-reconstruction pairs) and t-SNE visualizations;

    - Classification architectures used and results obtained using each encoder's weights;
    - Comparison and detailed analysis of the results obtained using the developed AE, VAE and DAE architectures.
  - **Part II**:
    - A detailed analysis of the results obtained;
    - A comparison between achieved results by training the YOLOv5 from scratch and by using transfer learning;
    - You should write any additional implementation details that you see fit;

- Google Colab notebooks (.ipynb);
- All modified files related to the YOLOv5.