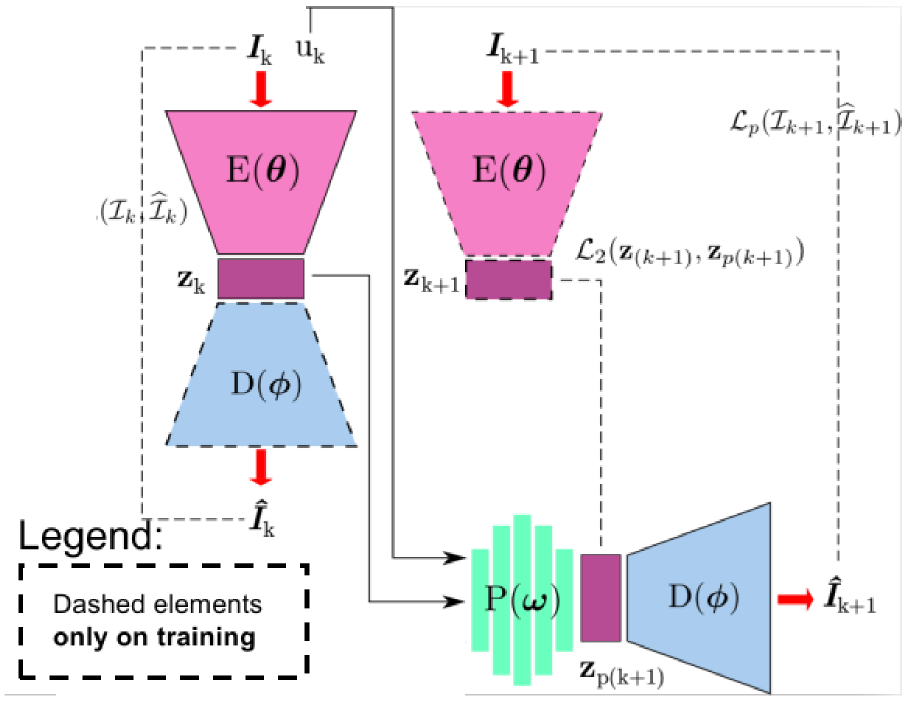
Machine Learning & Deep Learning
Advanced Image Analysis
MSCV VIBOT & ESIREM Robotique
Renato Martins - Universite de Bourgogne, September 2023

Assignment I - Learning to Sum and to Generate Images of Numbers

In this assignment, you will practice several concepts seen in the class, such as designing and manipulating MLPs, CNNs, autoencoders, image generation, generative models, and supervised classification. This practical is incremental with several intermediary steps to help you understand the development of the final model. You will design the complete final model such as to learn how to understand images of numbers, add images of numbers, and then displaying an image of the resulting number of the addition. An overview scheme of your final model is as follows:



where the encoder, decoder and Addition models are denoted by E, D and P respectively. Dashed modules and lines indicate the information flow used only during the training. You will again use the data from the simple (and widely used) MNIST digits dataset for this practical exercise. Before starting this practical, please have a look at the numerous tutorials, GitHub repositories, python scripts and notebooks available for developing simple classification models with MNIST discussed in the class lectures. Please also have a look on the slides of the class for details on PyTorch for training neural networks.

Part I - Environment Setup

Please create a new notebook on Colab named ML_Learning2Add, based on your previous notebooks, or follow these steps to configure your conda environment if you are working in your local machine:

```
>> conda create -n ML_Learning2Add
>> conda activate ML_Learning2Add
>> conda install jupyter
# Go your workspace directory of the course and run jupyter
>> jupyter-notebook
```

We will use mostly PyTorch tensors and Numpy arrays during this exercise. So do not hesitate to check some definitions on the tutorials on PyTorch and for Numpy. There are also some helpful "Cheat Sheets" of basic commands for Numpy and Scipy.

IMPORTANT: You should use TensorBoard or wandb[1] for tracking and monitoring your models.

Part II - Reconstruction Autoencoder

We will start by designing an autoencoder that learns to reconstruct the images of the numbers. For that, you will design the base encoder and decoder models:

1. Implement the encoder, as seen during the class, with the following specs:
   - Conv (32, 3x3) padding = 'same', ReLU , (Max-Pooling 2, stride 2)
   - Conv (64, 3x3) padding = 'same', ReLU , (Max-Pooling 2, stride 2)
   - Conv (128, 3x3) padding = 'same', ReLU , (Max-Pooling 2, stride 2)
   - MLP (256), Linear
   - MLP (2), Linear
2. Implement three possible versions of the decoder varying operations (as discussed during the class):
   - Exact feature mirror model from the encoder (including the reserve of max-pooling).
   - Using convolutions, activations, and upsampling.
   - Reshaping the input images to a 64×64 image and using upsampling.

   Which activation should be used in the output layer of your decoder model? Why?
3. How could you ensure the same conditions to train and evaluate your models, i.e., you would like that the same model trained twice on the same data to have similar performance (and similar weights)? This is useful for comparing versions of models. How could you do this?
   Tip: you can have a look at reproducibility and initialization with defined seeds on PyTorch to help your code to become deterministic[2]
4. Train your models using an L2 reconstruction loss for 100 epochs and Adam with a learning rate 0.001. Please provide the reconstruction error curves (using RMSE metric average for all pixels) for the training and validation data sets for your three models using TensorBoard or wandb. Which one is the best architecture and which one you would stick with? Why?
5. Do you observe any overfitting in the obtained training curves? How would you select the best weights over the epochs for a specific architecture?

[1]https://docs.wandb.ai/quickstart
[2]https://pytorch.org/docs/stable/notes/randomness.html
https://discuss.pytorch.org/t/random-seed-initialization/7854/21

6. Please retrain your models now using binary cross-entropy as the reconstruction loss. What do you observe compared when using L2? Why it is possible to use binary cross-entropy (BCE) with the data from MNIST?

7. Change the dimension of the code from 2 to 7 in your encoder (and operations in the decoder if needed). Please show visualizations for five reconstructed numbers for each loss.

Part III - Learning to Perform Addition from Images

You will now design a model that learns how to add an image of a number to a real number, and then generates the resulting number of the addition[3]. For instance given the inputs(image, number_to_add) the outputs of your model should be the following:

- Input('image of 5', 0) = 'image of 5', Input('image of 3', 2) = 'image of 5', Input('image of 1', 4) = 'image of 5'
- Input('image of 1', 2) = 'image of 3', Input('image of 2', 1) = 'image of 3', Input('image of 3', 0) = 'image of 3'
- Input('image of 4', 5) = 'image of 9', Input('image of 9', 0) = 'image of 9'

This model has three main components: encoder (E), decoder (D) and code addition model (P). The visualization of the full model is given in the initial figure of the assignment. You will use both the best encoder and decoder architectures you have done in the previous item. You will then need to create the data samples for training, then design the code addition model and plug all together.

1. The first thing we going to need to make is a data preprocessing to generate new samples that will make your task. We will restrict ourselves to sums that results in numbers of one digit, i.e., 1+8 and 9+0 are fine but 1+9 or 8+3 are not valid operations. For each resulting number we should create tuples (Input, Output) such as possible inputs combinations that results into a specific number:

   - Possible input combinations for the number 0: [(0, 0)]
   - ...
   - Possible input combinations for the number 4: [(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]
   - Possible input combinations for the number 5: [(0, 5), (1, 4), (2, 3), (3, 2), (4, 1), (5, 0)]
   - ...

   We provide the following initial code to help you to build your new samples:

```python
# function to create dataset with new samples for the addition of numbers
def transform_data_add(x_train, y_train, img_size = [32,32]):

    number_list = [0, 1, 2, 3, 4 , 5, 6, 7, 8, 9]
    value_add_list = []
    for number in number_list:
        value_add_list.append(np.arange(number + 1))

    img_in = [], img_out = [], value_in = [], samples_count = []

    for number, value_add in zip(number_list, value_add_list):
```

[3]We are selecting the addition operation, but you can make several other versions from the same problem with subtraction, multiplication or other operations. Please make your imagination free

```python
        train_add = []
        test_add = []
        samples_number = []
        # find the indexes of images related to as 0, 1, 2, ..., n
        # which sum can be upt to n
        for i in range(number + 1):
            index_train = (y_train == i)
            train_add.append(index_train)
        # find the max number of images (output is train_add[number])
        max_number_ele = sum(train_add[number])
        # find the indexes to of sum to find the number
        for index in np.nditer(value_add):
            input_imgs = x_train[train_add[index]]
            if (len(input_imgs) >= max_number_ele):
                input_imgs = input_imgs[:max_number_ele]
                output_imgs = x_train[train_add[number]]
            else:
                output_imgs = x_train[train_add[number]]
                output_imgs = output_imgs[:len(input_imgs)]
            img_in.append(input_imgs)
            # in case we dont add any value -- shuffle the inputs--outputs to avoid
                overfitting
            if (number - index == 0):
                random.shuffle(output_imgs)
            img_out.append(output_imgs)
            #value_in.append([number - index[0]] * len(output_imgs))
            value_in.append([number - index] * len(output_imgs))
            samples_number.append(len(output_imgs))
        samples_count.append(samples_number)

    print("Distribution of values: ", samples_count)
    print("Values to add: ", value_in)
    img_in = [cv.resize(item, tuple(img_size)) for sublist in img_in for item in
        sublist]
    value_in = [item for sublist in value_in for item in sublist]
    img_out = [cv.resize(item, tuple(img_size)) for sublist in img_out for item in
        sublist]
    return img_in, value_in, img_out
```

Please visualize some input and output samples to understand what is being done. Please display the distribution of the output samples. Is this created dataset balanced?

2. Following the depicted overview of the method, please implement the addition model. Your model should receive as inputs the code provided by the encoder ($\mathbf{z}_k$) and the number to be added (u_k) and should output a code of the resulting number $\mathbf{z}_{p(k+1)}$ (that will be used later by the decoder to display). Create your model with the following specs:

   - MLP (32), ReLU
   - MLP (16), ReLU
   - MLP (16), ReLU
   - MLP (7), Linear

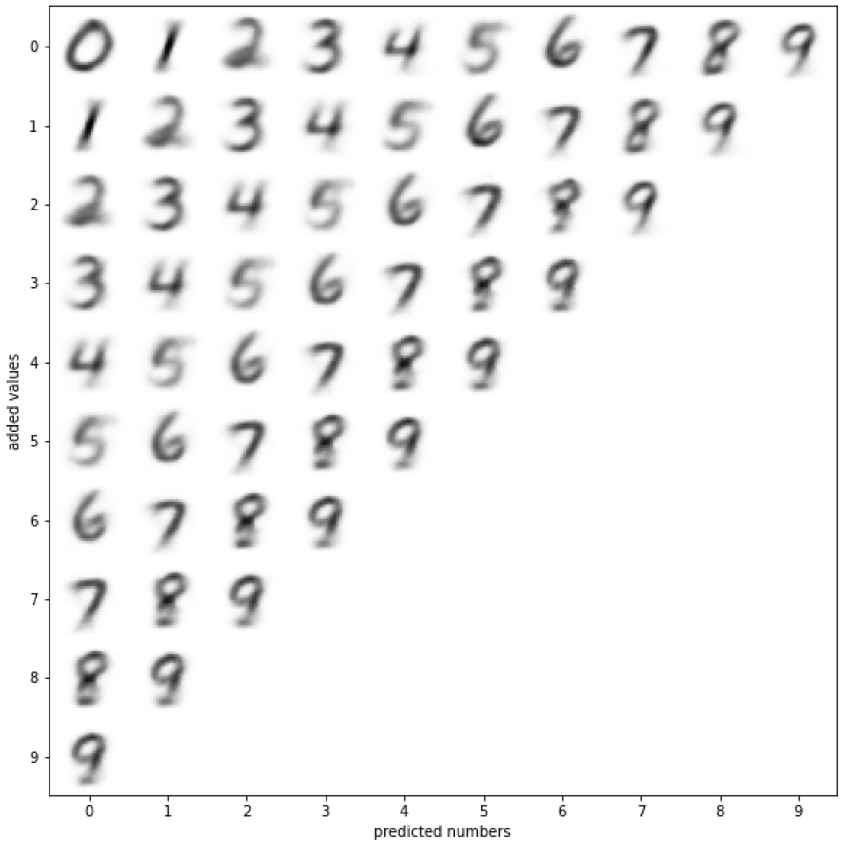
3. Now plug all the models together (following the scheme shown in the overview), taking care of instantiating the encoder and decoder twice (with shared weights between each instance). Your full training model should receive one image and the number to add, and provide two output images: the reconstruction of the input image and the expected output image from the addition.

4. Implement your training model loss, which is now the combination of three terms: i) Two reconstruction terms, one for the L2 error between the input encoding (as done in the previous item) and one L2 reconstruction loss between the generated number by your model and the expected one. Please use a scaling of 0.1 for the reconstruction of the input

in the loss.; ii) and a L2 reconstruction error between the codes generated by the addition model and the code resulting from the encoder of the output image (which is the one provided in the training sample). This term is to enforce the code to be similar.

5. Train your complete training model for 100 epochs using Adam with a learning rate of 0.001. Use as an evaluation metric the L2 reconstruction error between the expected added image and the one provided by the model. Was our model affected by overfitting?

6. Create an inference/test model using the learned encoder, decoder, and model addition submodels (that is only solid line modules should remain in the test as shown in the overview image). Your inference model should output only the expected number images of the addition. Use your inference model to generate the following visualization for some possible variations of the input images and numbers to be added. We provide an initial pseudocode for this visualization. You should obtain something similar to:



```python
def prediction_different_numbers(digit):
    max_value = 9 # np.max(np.array(digits.keys()))[0]
    predicted_images = []
    img_test_true = cv.resize(x_train_g[digits[digit]], tuple(img_size))
    for number in range(max_value - digit + 1):
        img_out = prediction(img_test_true, number)
        predicted_images.append(img_out)
    #return np.concatenate(np.array(predicted_images),axis=1)
    return np.concatenate(np.array(predicted_images),axis=0)

def display_images():
    # display 2D manifold of digits
    n = 10
    digit_size = img_size[0]
    figure = np.zeros((digit_size * n, digit_size * n))

    for index in range(n):
        images = prediction_different_numbers(index)
        figure[0:images.shape[0], index * digit_size: (index + 1) * digit_size] =
            images

    plt.figure(figsize=(n, n))
    start_range = digit_size // 2
    end_range = n * digit_size + start_range + 1
    pixel_range = np.arange(start_range, end_range, digit_size)
    plt.xticks(pixel_range, np.arange(10))
    plt.yticks(pixel_range, np.arange(10))
    plt.xlabel("predicted numbers")
    plt.ylabel("added values")
    plt.imshow(figure, cmap='Greys_r')
    #plt.savefig(pathresults + '/' + 'latent_vae_codesize_' + str(code_size) + '.
        png')
    plt.show()

display_images()
```

7. Finally, provide and visualize the generated output of your model to the following inputs: ('image of 7', 1), ('image of 7', 2) and ('image of 7', 1.5), ('image of 7', 1.4), ('image of 7', 1.6). What can you observe?

Part IV - Addition with Generative Model

Let's turn this model into a generative one, such as that we now handle distributions and generate new samples from it.

1. Please perform the required changes in the architecture for the encoder-decoder of your previous model to be a variational autoencoder (make the appropriate changes also in the losses). Do not forget the "reparameterization trick" and the respective KL losses.

2. Please re-train your variational model and perform the inference again for the following inputs : ('image of 7', 1), ('image of 7', 2) and ('image of 7', 1.5), ('image of 7', 1.4), ('image of 7', 1.6). Compare the generated results with the previous model in Part III. What can you observe?

3. Finally, go a step further and also make your MLP addition model (P) a generative one, i.e., it now also outputs the mean and standard deviation from the distribution. Compare the results against the two previous models.

Submission

Please return a concise PDF report explaining your reasoning and visualization of your results (as well as the discussion/answers to the questions). You should also submit your Python notebook (commented) with the corresponding implementations, together inside a [name]_DL_List1.zip file (replacing [name] with your name ;-)). The submission should be done via the Teams channel of the course using Teams assignment.
Deadline: 25/10/2023 at 23:59pm.

Note: This assignment is individual. Copying the work from another student will be awarded a 0 mark. In case there are multiple submissions with the same work, each one will receive a 0.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, "Deep Learning", MIT Press, 2016.