Exercise 3

Deadline: 06.12.2023 16:00.
Ask questions in discord to #ask-your-tutor

In this exercise, you will implement and test autoencoders.

Regulations

Please implement your solutions in form of *Jupyter notebooks* (`*.ipynb` files), which can mix executable code, figures and text in a single file. They can be created, edited, and executed in the web browser, in the stand-alone app JupyterLab, or in the cloud via Google Colab and similar services.

Create a Jupyter notebook `normalizing-flows.ipynb` for your solution and export the notebook to HTML as `normalizing-flows.html`. Zip all files into a single archive `ex02.zip` and upload this file to MaMPF before the given deadline.
Moreover, please set your **Anzeigename/display name** and **Name in Uebungsgruppen/name in tutorials** in MaMPF to your real name, which should be identical to your name in `muesli` and make sure you **join the submission** of your team via the invitation code before the submission deadline. Check out https://mampf.blog/handing-in-homework-assignments for instructions.

1 Two moons with an invertible neural network

We repeat task 2 from exercise 02 with a normalizing flow. That is, use a RealNVP network to train a generative model $p(x)$ for the two moons dataset. A RealNVP consists of coupling blocks, where the second half of the dimensions is transformed by a linear function whose coefficients depend on the first half. The coefficients are calculated with nested neural networks with two hidden layers (with ReLU activation) and a linear output layer (for the translation coefficient) resp. a linear layer followed by $\exp(\tanh(\tilde{s}))$ (for the scaling coefficient). The width of these internal networks is a hyperparameter and may be larger than the data dimension. The first half of the dimensions is just forwarded by a skip connection. After each coupling layer (except for the last), a random orthogonormal matrix is inserted to rotate the space, so that the skip connection is not always applied to the same dimensions. The RealNVP's constructor should have three arguments:

input_size: the data dimension (= 2 for the two moons dataset)

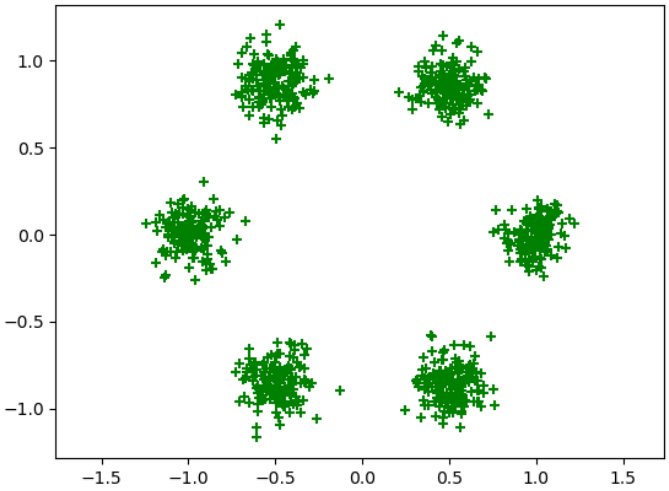
hidden_size: the width of the nested networks

blocks: the number of coupling blocks

Implement the encoder as the forward pass through the network, and the decoder as the reverse pass. Furthermore, implement a training function `train_inn()` (for "invertible neural network") using the `torch.optim.Adam` optimizer with default learning rate `lr=0.001` and the negative log-likelihood loss, as explained in the lecture. Training has again three hyperparameters: the size of the training set, the number of epochs, and the learning rate. During design and debugging of your code, your should start small (e.g. 1000 training points, 10 epochs, 2 layers) to ensure rapid training. Gradually increase network size and training effort once your results start looking promising. As a basic check, make sure that the reverse pass is the exact inverse of the forward pass, up to tiny numerical errors.

When your code works, train and evaluate the RealNVP with two moons at noise level 0.1. Investigate systematically the effect of the hyperparameters on model quality. Check if the code distribution is indeed standard normal. Note, however, that looking at reconstructions errors is pointless, because they are zero by design, as long as there is no bug. Furthermore, check the quality of the generated data. To this end, your RealNVP should have a function `RealNVP.sample(self, num_samples)` that generates the requested number of synthetic points. Report the MMD between a testset from `sklearn.datasets.make_moons` and generated points and show that visually better results correspond to smaller MMD. Comment on your observations.

Repeat the experiments (at good hyperparameter settings) with a Gaussian mixture whose centers are on the corners of a regular hexagon, like in the figure below. The standard deviations shall be 1/10 of the radius of the hexagon. Again comment on your observations – is the GMM more difficult than the two moons?



2 Two moons with a conditional invertible neural network

The function `sklearn.datasets.make_moons` also returns a label for the moon where each data point belongs to, and your function to generate the GMM on the hexagon can easily be extended to do this as well. We can use this information to train a conditional RealNVP for the conditional distribution $p(x \mid y)$ with y the label. Here, the nested networks receive a one-hot encoding of the label as an additional input for both the forward and reverse passes through the RealNVP. Extend the implementation from task 1 accordingly. The constructor needs a new argument `condition_size`, and the encoder and decoder functions must now accept a condition of appropriate size.

Make sure that the generalized design works in vectorized mode. Especially, the extended function `RealNVP.sample(self, num_samples, conditions)` should generate `num_sample` points for *every* row of `condition`.

Now, train the conditional INN with the labels from the two moons and the GMM. In addition, train for the GMM with only two labels, such that two peaks get label 0, and the other four are label 1 (you can imagine that two peaks are red, and the others are green). Evaluate your networks first in conditional mode $p(x \mid y)$, i.e. by evaluating for one label at a time, and compare with test data from the true conditional distributions.

Then, merge synthetic data from all labels and compare the marginal distributions $p(x)$. Is the quality of the conditional INN better or worse than the plain INN from task 1?

3 Higher-dimensional data with an INN

1. Repeat task 1 with the digits dataset (`sklearn.datasets.load_digits()`). Since these images have size 8×8, the codes will be 64-dimensional. Plot various 2D projections of the code space distribution for test data and check if the codes are indeed standard normally distributed.

Now generate data with your model. You will probably notice that the quality is not as good as what you got from an autoencoder in exercise 02. This is likely due to the fact that the dataset is too small to fully train a 64-dimensional code space. Recall, thet the code space in exercise 02 was at most 8-dimensional.

2. To reduce this problem, we can introduce an artificial bottleneck: In addition to the NLL loss, we also minimize a reconstruction loss. For this to make sense in the presence of perfect reconstruction, we decode the data only from the first k code dimensions (with k a hyperparameter, e.g. $k = 2, 4, 8$). The remaining code dimensions are set to zero before decoding and calculating the reconstruction loss (but not for the calculation of the NLL loss). This trick ensures that the important information about reconstruction must be encoded in the first k dimensions, like in a standard autoencoder. The remaining $64 - k$ code dimensions should contain the less important details.

   Now generate data with this model, with both the unimportant dimensions kept at zero (so only the first k dimensions are sampled), and with the first k dimensions kept at some fixed value and sampling the remaining $64 - k$ dimensions from the standad normal. The former variant should work similar to the autoencoder from exercise 02, whereas the latter should add diversity to the generated images, while keeping the general appearance fixed. Experiment with this split method, where codes are categorized into "important" and "unimportant" and report your findings. Does the quality of the synthetic images improve, and do you observe the expected behavior of the two sampling schemes described above?

3. Another possibility is to use more training data. The listing below shows how you can import the MNIST digits dataset (which consists of 6000 training instances for each label) and downscale it to 8×8 on the fly. Evaluate your RealNVP on this dataset.

```
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from PIL import Image, ImageFilter  # install 'pillow' to get PIL

import matplotlib.pyplot as plt

# define a functor to downsample images
class DownsampleTransform:
    def __init__(self, target_shape, algorithm=Image.Resampling.LANCZOS):
        self.width, self.height = target_shape
        self.algorithm = algorithm

    def __call__(self, img):
        img = img.resize((self.width+2, self.height+2), self.algorithm)
        img = img.crop((1, 1, self.width+1, self.height+1))
        return img

# concatenate a few transforms
transform = transforms.Compose([
    DownsampleTransform(target_shape=(8,8)),
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor()
])

# download MNIST
mnist_dataset = datasets.MNIST(root='./data', train=True,
                               transform=transform, download=True)

# create a DataLoader that serves minibatches of size 100
data_loader = DataLoader(mnist_dataset, batch_size=100, shuffle=True)

# visualize the first batch of downsampled MNIST images
def show_first_batch(data_loader):
    for batch in data_loader:
        x, y = batch
        fig = plt.figure(figsize=(10, 10))
    for i, img in enumerate(x):
        ax = fig.add_subplot(10, 10, i+1)
        ax.imshow(img.reshape(8, 8), cmap='gray')
        ax.axis('off')
    break
```

4 Higher-dimensional data with a conditional INN

Repeat task 3 with a conditional RealNVP using the digit labels as a condition. Generate data with your model for predefined labels. Do the synthetic digits look like examples for the desired digit label? If yes, you should again check the accuracy and confidence with a random forest classifier as in exercise 02.