Project 1: Variational Autoencoder

- ▷ Projects should be submitted in teams of 3-4 students.
- ▷ Accepted format is a zip file containing the Python scripts/notebooks, the results and a report (1 A4 page - font size 12) of your work. Please put the full names and student ID numbers of all team members on the report.
- ▷ Submissions are made via a Sciebo dropoff link: `https://fz-juelich.sciebo.de/s/3u2tF9ynptGEOqm`.
- ▷ Submissions have to follow the naming scheme *project01_USER1_USER2_USER3_USER4.zip* using your university username ("Benutzername" in ILIAS/IDM). If a second version is uploaded use the ending *_v2.zip* for the file. Only the latest version will be rated. Make sure you name the zipped folder the same as the zip file.
- ▷ The project is either a success or a failure.

The olfactory tubercle is a brain region in the basal forebrain that receives direct input from the olfactory bulb, a neural structure involved in olfaction (smelling). The olfactory tubercle contains *terminal islands*, small cell clusters containing small cells of varying shape and size. Terminal islands are clusters of small cells scattered among the intermingling nuclei of the basal forebrain. There are two main morphological types of terminal islands: *granocellular islands* Fig. 1 characterized by small, round, densely packed neurons and *parvicellular islands* Fig. 2 which consist of small neurons with somewhat larger and less ovoid cell bodies than in the granocellular islands. Thus, it is assumed that terminal island contain progenitor cells ceased in different stages of development. However, although linked to pathophysiology of schizophrenia their functions remain largely unknown. Therefore, in order to set ground floor for future neuroimaging studies, it is important to gain insight into 3D characteristics of the terminal islands and their location and their extent. Terminal islands were annotated in different brains, which were cut into thin sections and stained for cell bodies. However, the available annotations do not distinguish terminal islands containing different cell types. The goal of this project is to automatically identify structural differences in terminal islands. All deep learning implementations should use PyTorch. Tasks marked with *Advanced* are aimed at ambitious participants of the lecture and are not mandatory.

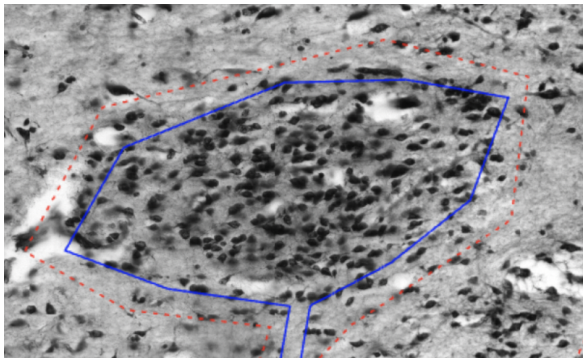


Figure 1: A granucellular terminal island in B20.

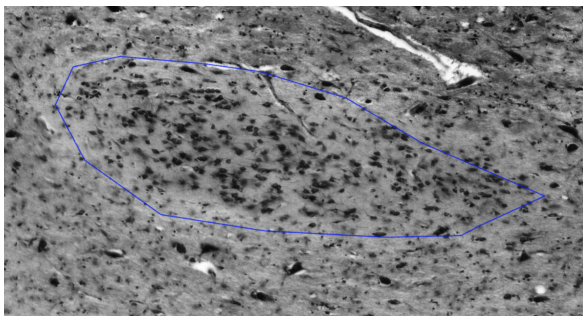
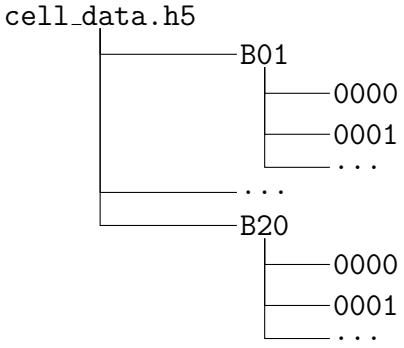


Figure 2: A parvocellular terminal island in B20.

1. **Data Exploration**

   Download the dataset from this link. The dataset consists of one large file (2.5 GB) in HDF5 format. This data format is very convenient to use with large scale microscopic data. To open the file and read the data you might want to use h5py. The structure of the content is as follows:



   with HDF5 groups B01, B02, B05, B07 and B20, which each correspond to a different brain. Within each group each entry then corresponds to a single image as HDF5 dataset.

   (a) Plot the number of images per brain.

   (b) Plot the distribution of image sizes in an interpretable way.

   (c) Checkout the largest and smallest images per brain and visualize them.

   (d) Based on your data exploration decide for one test brain group and keep the other groups for training.

   (e) Determine and store the global `maximum`, `minimum`, `mean` and `standard deviation` for the image pixel values of the training data.

2. **Sampling Strategy**

   The large extent of microscopic images and their varying sizes pose a problem for deep learning models that usually require small and fixed sized inputs. One strategy to process such images anyways is by dividing them first into a grid of equally spaced tiles that cover the whole image plane (pad the borders if necessary).
   To maximize the variance in training examples it is common practice to sample batches of tiles from random locations and random images.

   (a) Write a PyTorch Dataset that stores the training images (or a subset of them for debugging) to RAM. Overwrite the `__getitem__` method such that it takes a tuple of `(brain, image, row, column, tile_size)` as input and extracts a tile of size `(1, tile_size, tile_size)` from dataset `image` in group `brain` at location `(row, column)`. Note that the first axis corresponds to the PyTorch channel dimension that we need to add here since all images are gray-value. Convert the tile to a `torch.Tensor` of type `float32` and standardize it by your determined global `mean` and `standard deviation` before returning it.

   (b) Write a PyTorch Sampler to sample random training tile locations. Overwrite the `__iter__` method such that it yields random tuples of `(brain, image, row,`

`column, tile_size)` based on your train brains, their image shapes and the selected `tile_size`. Note that you need to make sure here that the tiles stay within the boundary of each image. Overwriting the `__len__` method sets the number of example tiles you draw per epoch.

(c) Create a PyTorch DataLoader object that takes the `Dataset` and `Sampler` as input. Use a `tile_size` of 64 and a `batch size` of 8 to visualize a single batch.

Tip You should scale your data back to the original `minimum`, `maximum` range in order to visualize them.

Tip Do not use `np.random` for drawing random numbers in your `Sampler`. There are known issues with PyTorch workers sharing the same seed. Use the python builtin `random` or `torch.random` instead.

Advanced Assigning to each image an equal probability to be selected for training might not be the best choice to create i.i.d. training data. Can you imagine a better sampling strategy?

Advanced You might want to experiment with several data augmentations in your training `Dataset`.

3. **Variational Autoencoder**

Implement a variational autoencoder model that compresses each tile down to a low dimensional feature representation.

(a) Implement your own idea of a variational autoencoder as PyTorch Module or get some inspiration from the internet. Your variational autoencoder should be able to compress the tiles down to a feature space of 128 dimensions. The model should consist of an independent `Encoder` and `Decoder`.

(b) Decide for an optimizer. A good baseline is to use ADAM with a learning rate of 0.001.

(c) Implement the training loop that iterates over your `DataLoader`, passes the batch through the variational autoencoder model and backpropagates the loss composed of a weighted sum of a `reconstruction loss` (MSE between the original batch and the reconstructed one), the `KL-divergence loss` (quantification of the difference between the learned latent distribution and a standard normal distribution) and a `regularization loss` (e.g. L2 or L1 of your model paramters). You can start with a small weight for the regularization loss (e.g. 1e-6) and experiment with different setups later. An example training loop can be found here

Tip Make sure to reduce the spatial size with several convolutions before applying any fully connected layers to reduce the number of parameters in your VAE.

Tip Adjust the number of workers in your `DataLoader` to your number of available CPUs.

Advanced If you have access to more powerful hardware you can increase the `tile_size` and `batch size` for your model (e.g. `tile_size`=128 and `batch size`=64)

Advanced You can experiment with different feature space sizes. What happens in the extreme cases of 2 or 4096?

4. **Model training**

   (a) Check the initial loss for your pipeline and assess whether each component is plausible.

   (b) Debug your training pipeline by training one epoch on a reduced training dataset (e.g. only two images per brain) until you observe a falling loss.

   (c) Train your model until convergence of the loss and store the paramters in a PyTorch State Dict.

   Tip Steps 1 and 2 should be possible on your local machine and in notebooks. For step 3 you should rewrite your code into an executable python script and switch to GPU accelerated hardware. Google Colab allows access to GPUs free of charge if you prefer working with notebooks.

   Advanced You might want to attach a tensorboard SummaryWriter to your model training to monitor your loss.

   Advanced If you have access to more powerful hardware you can get deeper with the model and higher with the `tile_size`. Experiment with different setups.

5. **Feature analysis**

   (a) Sample 10 random latent representations from a standard Gaussian distribution in the latent space of your trained `Variational Autoencoder`. Decode each latent vector using the decoder to generate corresponding images. Discuss how well the Variational Autoencoder can synthesize plausible brain structures.

   (b) Select two tiles from different regions of the test brain and encode them with your trained Variational Autoencoder to obtain their latent representations. Define k interpolation steps and linearly interpolate between these vector s. Decode each interpolated vector to generate intermediate images, and analyze how the reconstructions transition between the structural and textural features of the original tiles. Repeat this process for two types of tile pairs: tiles within the same section and tiles from different sections.

   (c) Sample ~ 3.000 tiles from the `test` brain. Pass them through your trained `Variational Autoencoder` (`Encoder`) to generate latent space representations. Reduce the dimensionality of the representations using Principal Component Analysis (PCA) or Uniform Manifold Approximation and Projection (UMAP) to obtain a 2D visualization. Plot the reduced latent space and examine the distribution of the encoded tiles. Discuss whether the latent space appears smooth and interpretable.

   Tip Remember to use the same standardization parameters as for the training.