imaging_ai

Kasra Eskandarizanjani Mina Naseh Paria Ighanian

```
import sys
sys.executable
```

```
'/opt/hostedtoolcache/Python/3.10.16/x64/bin/python3'
```

---

title: Task 1

Task 1

The Fourier transformation $f(x,y) \rightarrowtail F(u,v)$ of a greyscale image $f(x,y)$ results in a band-limited signal in the spatial frequency range with maximum frequencies f_{umax} and $_{vmax}$. For representation in the computer, the (partial) image is sampled in x direction with 20 sampling points per mm and in y direction with 10 sampling points per mm.

1. What is the theoretical maximum value of f_{umax} and f_{vmax} if error-free image reconstruction from the digital image should be possible (not using any compressive-sensing techniques)? (6pts)

According to the Nyquist sampling theorem, the maximum representable frequency (Nyquist frequency) in each direction is half the sampling frequency. The sampling frequency can be derived from the given sampling points per mm.

- Sampling frequency in x is f_{sx} and the Nyquist frequency in x is f_{umax}:

$$f_{sx} = 20 \text{ points/mm} = 20 \times 10^3 \text{ points/m}$$
$$\implies f_{umax} = \frac{f_{sx}}{2} = 10.0 \text{ cycles/mm}$$

- Sampling frequency in y is f_{sy} and the Nyquist frequency in y is f_{vmax}:

$$f_{sy} = 10\,\text{points/mm} = 10 \times 10^3\,\text{points/m}$$
$$\implies f_{vmax} = \frac{f_{sy}}{2} = 5.0\,\text{cycles/mm}$$

This ensures error-free reconstruction, as the digital image will contain all frequency components of the original image within the Nyquist limit. Frequencies above these limits would result in aliasing, violating error-free reconstruction conditions.

What is the minimum memory requirement for the color image $f_F(x, y)$ when stored in a conventional computer system, if 1024 values are to be distinguished per color channel. Describe the image format to be used.

To start lets find the number of ixels

Let the image dimensions in mm be L_x (width) and L_y (height).
- Pixels in x-direction: $N_x = 10.0 \cdot L_x$ - Pixels in y-direction: $N_y = 5.0 \cdot L_y$ - Total number of pixels:

$$N_{\text{pixels}} = N_x \cdot N_y = 50.0 \cdot L_x \cdot L_y$$

Each pixel in a color image has values for three color channels: Red, Green, and Blue (RGB). Each channel can store 1024 distinct values, which means $log_2^{1024} = 10.0$ bits per channel.

Total bits per pixel: $b = 10.0 \times 3 = 30.0$ bits/pixel.

The memory requirement is the product of the number of pixels and bits per pixel:

$$\text{Used Memory} = N_{\text{pixels}} \cdot b = (50.0 \cdot L_x \cdot L_y) \cdot b\,\text{bits} = 6.25 \cdot L_x \cdot L_y \cdot 30.0\,\text{bytes} = 187.5 \cdot L_x \cdot L_y\,\text{bytes}$$

How many colors could be represented with the quantization chosen in sub-task 3? (2pts)

Each channel (Red, Green, and Blue) can represent 1024 intensity levels. With 10 bits per channel and 3 channels, the total number of colors is:

$$\text{Total colors} = 1024^3 = 1,073,741,824$$

---

title: “task 2”

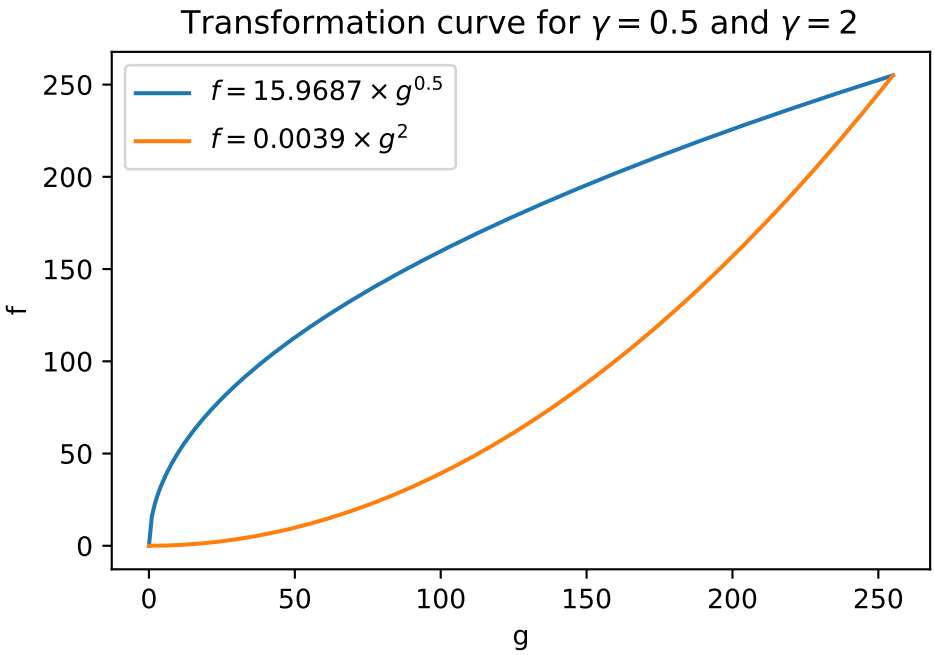
Task 2

For the subjective enhancement of a greyscale image $G = g(x, y)$, a transformation T_G is performed as a so-called gamma correction in the form $T_G : g \to f$ with $f(x, y) = cg^{\gamma}(x, y)$ where $g, f \in [0, 255]$.

Sketch the transformation curve T_G for $\gamma_1 = 0.5$ and $\gamma_2 = 2$

The first step is to find the values of c for both cases. Since $\max(f) = \max(g) = 255$, we have $c = 255/255^{\gamma}$.

```
from matplotlib import pyplot as plt
import numpy as np


def draw_transform_curve(gamma: float, ax: plt.Axes = None, label: bool = True):
    if not ax:
        fig, ax = plt.subplots()
    x = np.linspace(0, 255, 256)
    c = 255 / 255**gamma
    y = c * x**gamma
    message = f"$f = {c:0.4f} \\times g^{{{gamma}}}$"
    if label:
        ax.plot(x, y, label=message)
    else:
        ax.plot(x, y)
    ax.set_xlabel("g")
    if label:
        ax.set_ylabel("f")
    else:
        ax.set_ylabel(message)


fig, ax = plt.subplots()
for gamma in [0.5, 2]:
    draw_transform_curve(gamma, ax)
ax.set_title(f"Transformation curve for $\\gamma=0.5$ and $\\gamma=2$")
ax.legend()
plt.show()
```



How is the coeficient c typically determined? (2pts)

The coefficient c is typically determined such that the maximum value of the input image is mapped to the maximum value of the output image. This is done to ensure that the full dynamic range of the output image is used.

As mentioned above, $c = 255/255^{\gamma}$.

In which respect and for which type of input images G do the two gamma values γ_1, γ_2 lead to an image enhancement respectively? (2pts)

For $\gamma < 1$, the transformation curve is concave, which means that the lower intensity values are stretched more than the higher intensity values. This leads to a brighter image with more contrast. This is useful for images with low contrast.

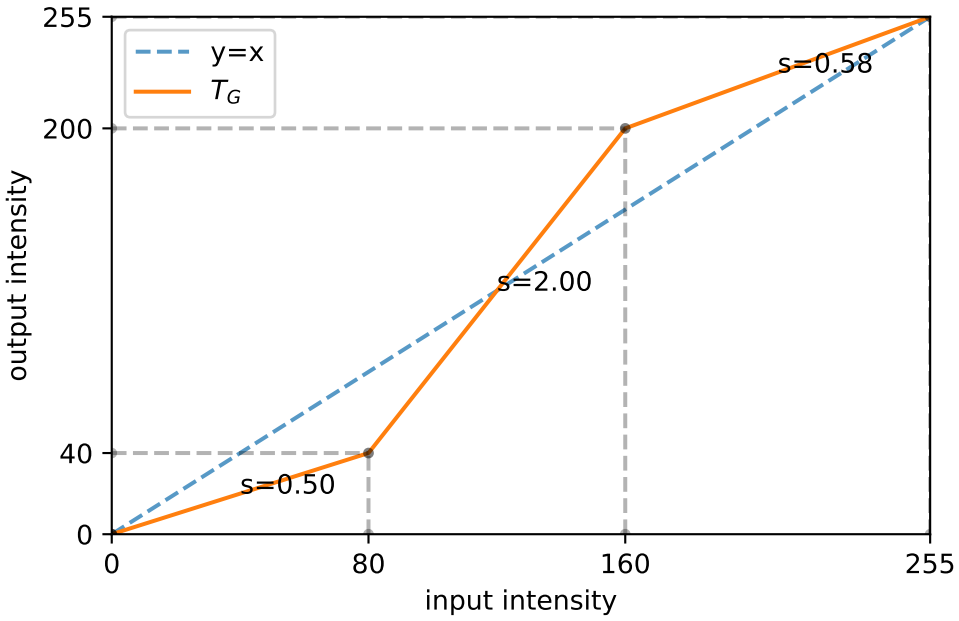
For $\gamma > 1$, the transformation curve is convex, which means that the higher intensity values are stretched more than the lower intensity values. This leads to a darker image with more contrast. This is useful for images with high contrast.

What should be the minimum slope of the transform function?

1. for a grey value spread (2pts)
2. for a grey value compression (2pts)

It's important to note that a slope of exactly 1 implies no change in contrast, as the transformation function becomes an identity mapping. Also, a slope of 0 implies that the output image will be a constant value, which is not useful for image enhancement.

1. For a grey value spread, the minimum slope of the transform function should be 1.
2. For a grey value compression, the minimum slope of the transform function should be 0 (and smaller than 1). For instance, in this function:



As we can see, the gray values between `{python} spread_range[0]` are streched between `{python} spread_range[1]` which has a slope greater than 1. On the other hand, the gray values between `{python} compress_range[0]` are compressed between `{python} compress_range[1]` which has a slope smaller than 1.

---

title: "task 3"

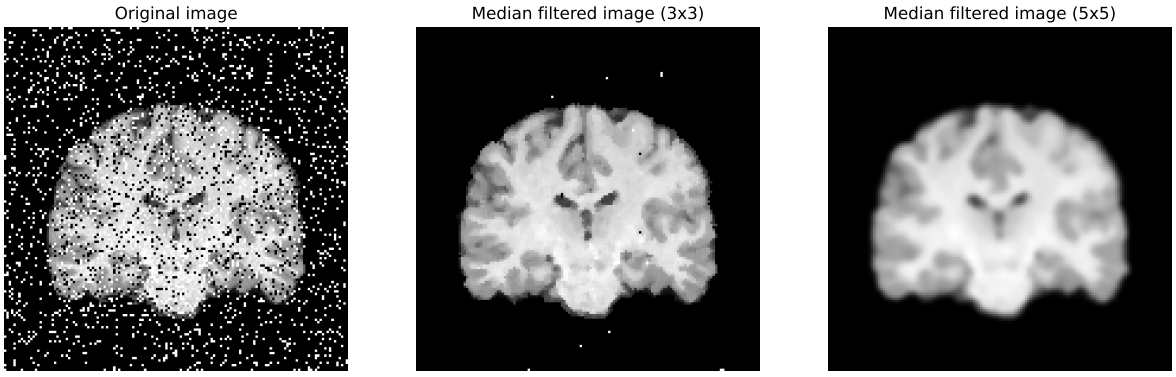
Task 3

In this task you will need to perform threshold-based image analysis:

Read the greyscale image brain.png, which is provided on the lecture homepage. Reduce the salt and pepper noise in the image using a median filter. (3pts)

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import os


img_noise = cv2.imread("brain-noisy.png", cv2.IMREAD_GRAYSCALE)
if img_noise is None:
    img_noise = cv2.imread("./reports/brain-noisy.png", cv2.IMREAD_GRAYSCALE)
assert img_noise is not None, "Image not found {}".format(os.listdir())
img = cv2.medianBlur(img_noise, 5)
img = cv2.GaussianBlur(img, (5, 5), 0)

fig, ax = plt.subplots(1, 3, figsize=(15, 5))
ax[0].imshow(img_noise, cmap="gray")
ax[0].set_title("Original image")
ax[0].axis("off")
ax[1].imshow(cv2.medianBlur(img_noise, 3), cmap="gray")
ax[1].set_title("Median filtered image (3x3)")
ax[1].axis("off")
ax[2].imshow(img, cmap="gray")
ax[2].set_title("Median filtered image (5x5)")
ax[2].axis("off")

plt.show()
```

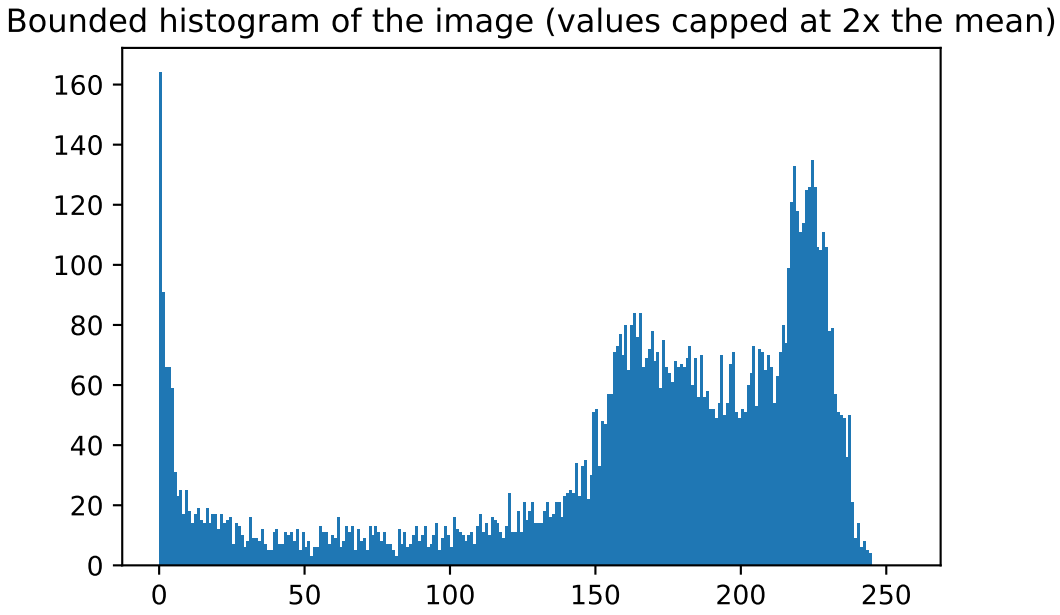
```
[ WARN:0@0.009] global loadsave.cpp:241 findDecoder imread_('brain-noisy.png'): can't open/re
```



As we can see the kernel size of 3×3 is not enough to remove the noise, while the kernel size of 5×5 is sufficient.

Otsu thresholding is a histogram-based method for image segmentation. Use it to find an intensity threshold to segment brain pixels from background. Use Otsu thresholding again to find the threshold only over the brain pixels to segment brain's grey matter from the white matter. Using the two thresholds create three binary masks brain-bg.png, brain-gm.png, brain-wm.png, which should be white in regions of background, grey matter, and white matter, respectively, and black elsewhere. (4pts)

```
values, bin_edge = np.histogram(img, bins=256, range=(0, 256))
bin_centers = (bin_edge[:-1] + bin_edge[1:]) / 2
# values = values[1:]
# bin_centers = bin_centers[1:]
m = values.mean() * 2
values[values > m] = m

plt.bar(bin_centers, values, lw=2)
plt.title("Bounded histogram of the image (values capped at 2x the mean)")
plt.show()
```

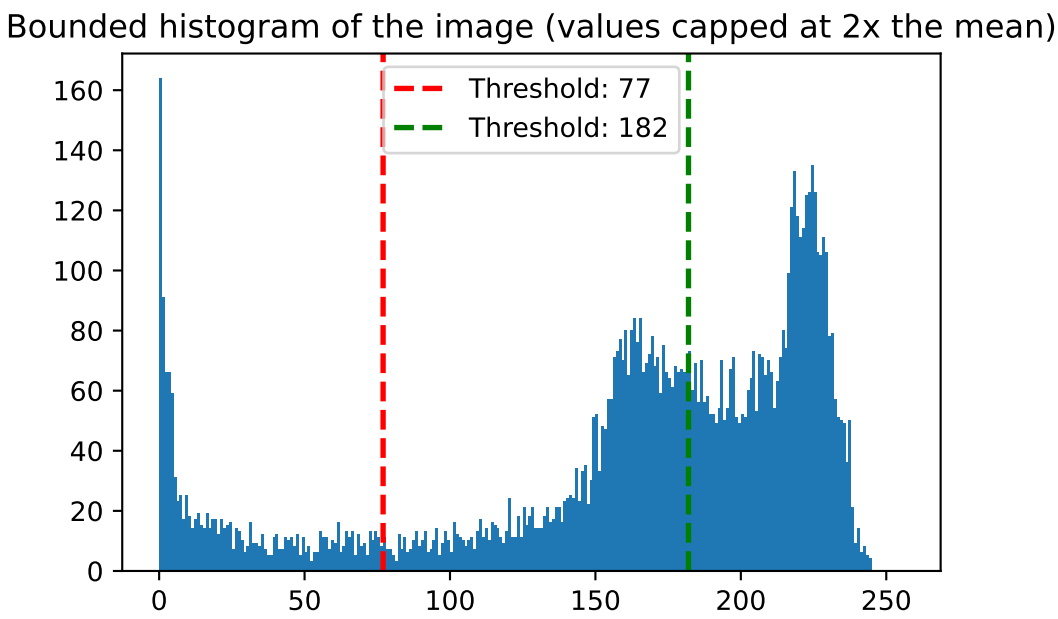


The correct way to use Otsu thresholding with several values is to use (Arora et al. 2008), which is not implemented in OpenCV. However, we can use the implementation in the `skimage` library (which implemented based on (Liao et al. 2001))

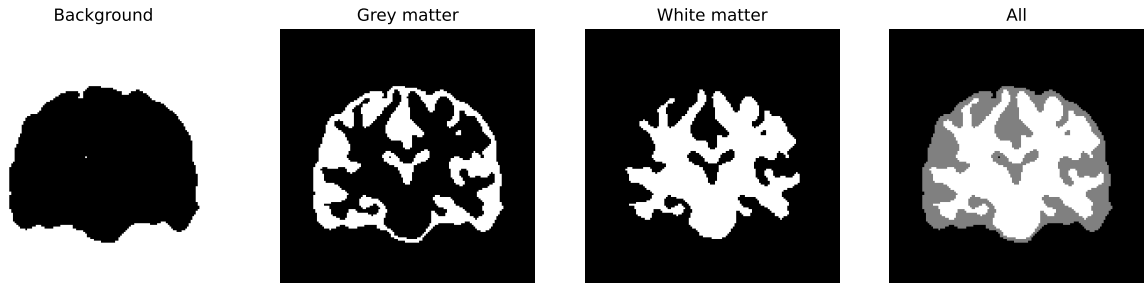
```python
from skimage.filters import threshold_multiotsu


def otsu_threshold(
    img: np.ndarray, classes: int
) -> tuple[list[np.ndarray], np.ndarray]:
    threshold = threshold_multiotsu(img, classes=classes).tolist()
    threshold = [0] + threshold + [255]
    assert (
        len(threshold) == classes + 1
    ), "The number of thresholds should be equal to the number of classes - 1"
    masks = [(img >= t1) & (img < t2) for t1, t2 in zip(threshold, threshold[1:])]
    # masks.append(img >= threshold[-1])
    assert all(mask.dtype == bool for mask in masks), "Masks should be boolean"
    assert (
        len(masks) == classes
    ), "The number of masks should be equal to the number of classes"
    return masks, threshold[1:-1]


(brain_bg, brain_gm, brain_wm), threshold = otsu_threshold(img, 3)
```

```python
colors = ["r", "g", "y"]
(brain_bg, brain_gm, brain_wm), threshold = otsu_threshold(img, 3)

print(f"Threshold for the whole image: {threshold}")

values, bin_edge = np.histogram(img, bins=256, range=(0, 256))
bin_centers = (bin_edge[:-1] + bin_edge[1:]) / 2
m = values.mean() * 2
values[values > m] = m

plt.bar(bin_centers, values, lw=2)
for th, color in zip(threshold, colors):
    plt.axvline(th, color=color, lw=2, ls="--", label=f"Threshold: {th}")
plt.legend()
plt.title("Bounded histogram of the image (values capped at 2x the mean)")
plt.show()
```

```
Threshold for the whole image: [77, 182]
```

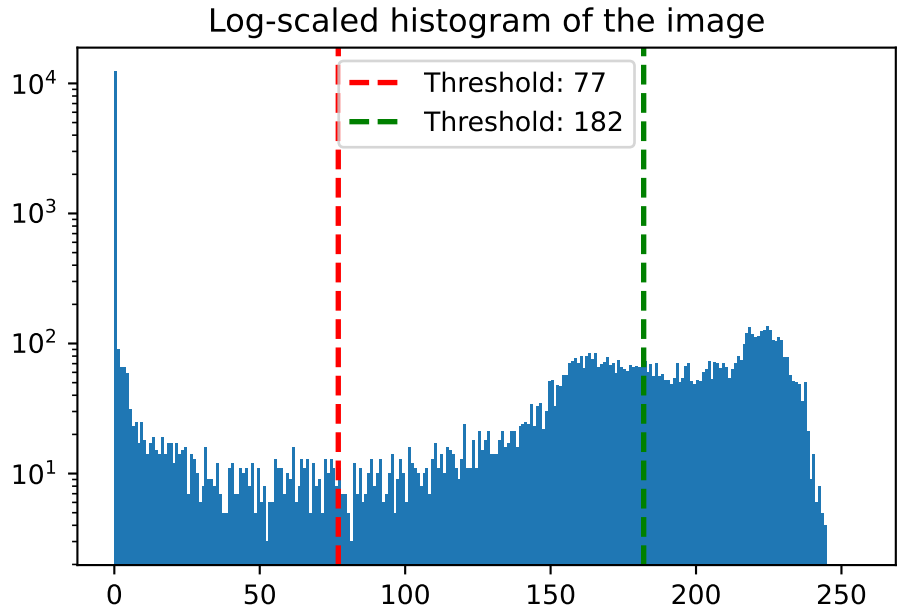



```
fig, ax = plt.subplots(1, 4, figsize=(15, 5))
ax[0].imshow(brain_bg, cmap="gray")
ax[0].set_title("Background")
ax[0].axis("off")

ax[1].imshow(brain_gm, cmap="gray")
ax[1].set_title("Grey matter")
ax[1].axis("off")

ax[2].imshow(brain_wm, cmap="gray")
ax[2].set_title("White matter")
ax[2].axis("off")

ax[3].imshow(brain_bg * 1 + brain_gm * 2 + brain_wm * 3, cmap="gray")
ax[3].set_title("All")
ax[3].axis("off")

plt.show()
```



Plot a log-scaled histogram of the image, which should show how frequently different intensity values occur in the image. How could you roughly estimate the two thresholds you found in the previous task just by looking at the histogram? (3pts)

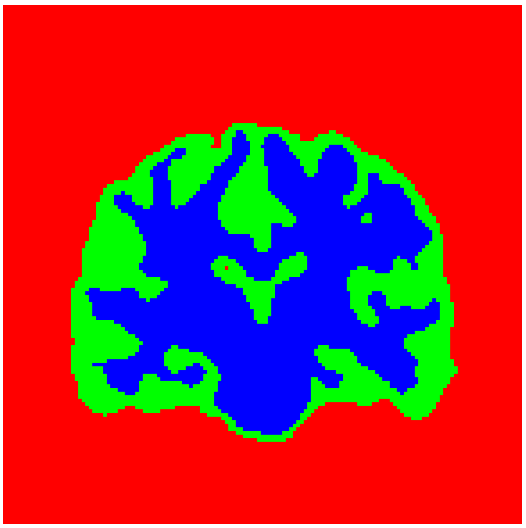
```
values, bin_edge = np.histogram(img, bins=256, range=(0, 256))
bin_centers = (bin_edge[:-1] + bin_edge[1:]) / 2
plt.bar(bin_centers, values, lw=2)
plt.yscale("log")

for th, color in zip(threshold, colors):
    plt.axvline(th, color=color, lw=2, ls="--", label=f"Threshold: {th}")

plt.legend()
plt.title("Log-scaled histogram of the image")
plt.show()
```



As we can see, the histogram has two peaks, which correspond to the grey matter and white matter. The two thresholds can be estimated by finding the two peaks in the histogram. (The purpose of otsu thresholding is to find the optimal threshold for the two peaks)

Combine the three masks into a single colour image so that background, grey matter, and white matter are mapped to red, green and blue, respectively. (3pts)

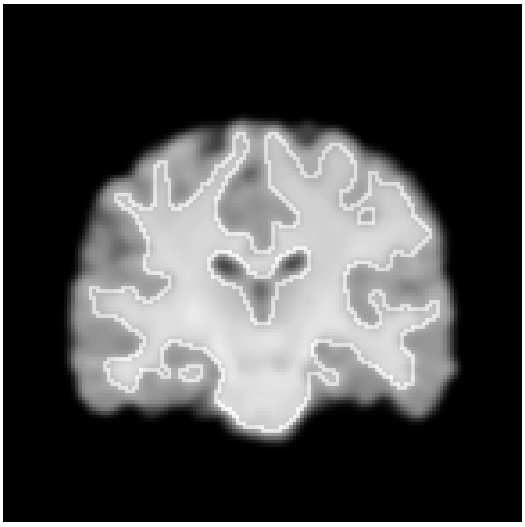
```
combined_brain = np.stack([brain_bg, brain_gm, brain_wm], axis=-1).astype(np.uint8) * 255

plt.imshow(combined_brain)
plt.axis("off")
plt.show()
```



Use erosion (or any other morphological) filter to produce a border between the grey and white matter. Overlay that border on the denoised input image. (3pts)

```
kernel = np.ones((3, 3), np.uint8)
brain_wm_eroded = cv2.erode(brain_wm.astype(np.uint8), kernel, iterations=1)
brain_wm_dilated = cv2.dilate(brain_wm_eroded, kernel, iterations=1)
border = (brain_wm_dilated - brain_wm_eroded) * 255
alpha = 0.85
bordered_img = cv2.addWeighted(img, alpha, border, 1 - alpha, 0)


# plt.imshow(img, cmap="gray")
# plt.imshow(border, cmap="gray", alpha=0.5)
plt.imshow(bordered_img, cmap="gray")
```

```
plt.axis("off")
plt.show()
```

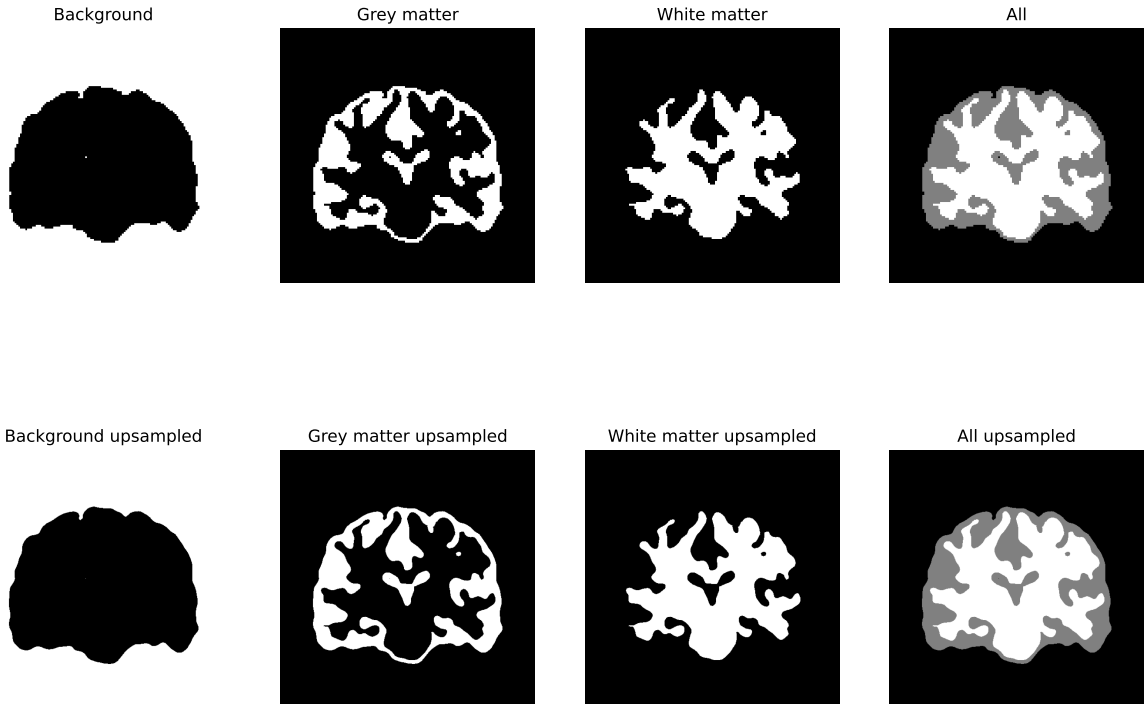


Use bilinear interpolation to up-sample the image by a factor of four along each axis. Apply the same thresholds as in 2) to obtain a segmentation into background, grey matter, and white matter. Up-sample the masks from 2) in the same way and compare the up-sampled masks to the masks from the up-sampled image. Can you see a diference? Why? Repeat the same procedure using nearest neighbour interpolation. Can you see a diference now? (4pts)

```
def upsample(img: np.ndarray, factor: int, interpolation: int) -> np.ndarray:
    return cv2.resize(
        img, (img.shape[1] * factor, img.shape[0] * factor), interpolation=interpolation
    )


masks, threshold = otsu_threshold(img, 3)
img_upsampled = upsample(img, 4, cv2.INTER_LINEAR)
masks_upsampled, threshold_upsampled = otsu_threshold(img_upsampled, 3)

fig, ax = plt.subplots(2, 4, figsize=(15, 10))
fig.suptitle(
    "Comparison of upsampled masks and upsampled image using linear interpolation",
    fontsize=16,
)

titles = ["Background", "Grey matter", "White matter", "All"]
```

```
masks.append(masks[0] * 1 + masks[1] * 2 + masks[2] * 3)
masks_upsampled.append(
    masks_upsampled[0] * 1 + masks_upsampled[1] * 2 + masks_upsampled[2] * 3
)

for i, (mask, mask_upsampled, title) in enumerate(zip(masks, masks_upsampled, titles)):
    ax[0, i].imshow(mask, cmap="gray")
    ax[0, i].set_title(title)
    ax[0, i].axis("off")

    ax[1, i].imshow(mask_upsampled, cmap="gray")
    ax[1, i].set_title(f"{title} upsampled")
    ax[1, i].axis("off")

plt.show()
```

Comparison of upsampled masks and upsampled image using linear interpolation



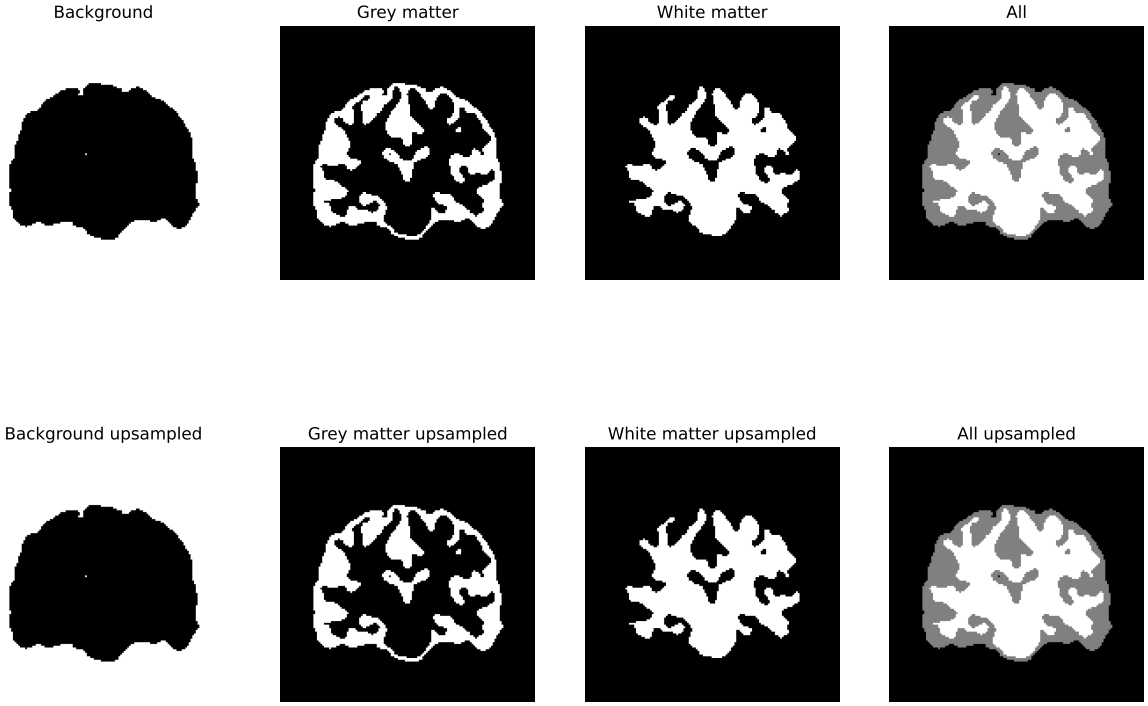
Clearly, we can see much smoother edges in the upsampled masks compared to the upsampled image. This is because the interpolation method used in the up-sampling process is linear,

which smooths the edges.

Now, let's repeat the same procedure using the nearest neighbour interpolation method.

```
masks, threshold = otsu_threshold(img, 3)
img_upsampled = upsample(img, 4, cv2.INTER_NEAREST)
masks_upsampled, threshold_upsampled = otsu_threshold(img_upsampled, 3)

fig, ax = plt.subplots(2, 4, figsize=(15, 10))
fig.suptitle(
    "Comparison of upsampled masks and upsampled image using nearest neighbour interpolation"
    fontsize=16,
)

titles = ["Background", "Grey matter", "White matter", "All"]
masks.append(masks[0] * 1 + masks[1] * 2 + masks[2] * 3)
masks_upsampled.append(
    masks_upsampled[0] * 1 + masks_upsampled[1] * 2 + masks_upsampled[2] * 3
)

for i, (mask, mask_upsampled, title) in enumerate(zip(masks, masks_upsampled, titles)):
    ax[0, i].imshow(mask, cmap="gray")
    ax[0, i].set_title(title)
    ax[0, i].axis("off")

    ax[1, i].imshow(mask_upsampled, cmap="gray")
    ax[1, i].set_title(f"{title} upsampled")
    ax[1, i].axis("off")

plt.show()
```

Comparison of upsampled masks and upsampled image using nearest neighbour interpolation



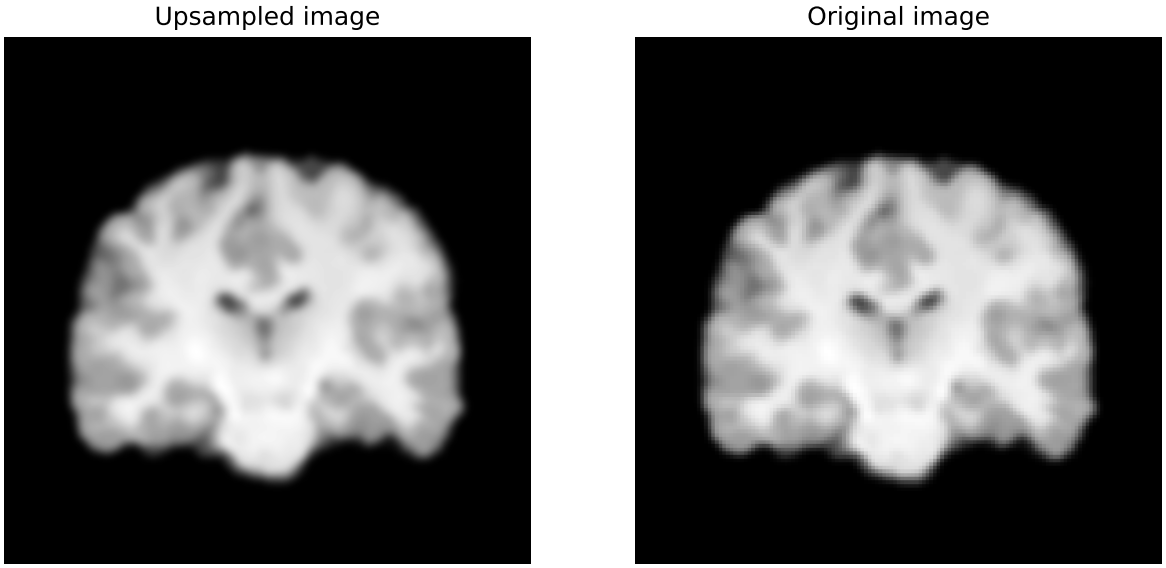
We can see the edges are much sharper in the upsampled masks compared to the upsampled image. This is because the nearest neighbour interpolation method does not smooth the edges.

TODO: Test same thing with pyrUp

```
def upsample_pyramid_(img: np.ndarray) -> np.ndarray:
    # return cv2.resize(
    #     img, (img.shape[1] * factor, img.shape[0] * factor), interpolation=interpolation
    # )
    return cv2.pyrUp(img, dstsize=(img.shape[1] * 2, img.shape[0] * 2))

def upsample_pyramid(img: np.ndarray, factor: int) -> np.ndarray:
    if factor <= 1:
        raise ValueError("Factor should be greater than 1")
    f = 1
    while f < factor:
        img = upsample_pyramid_(img)
        f *= 2
```

```
    return img


print(img.shape)
img_upsampled = upsample_pyramid(img, 4)
print(img_upsampled.shape)
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].imshow(img_upsampled, cmap="gray")
ax[0].set_title("Upsampled image")
ax[0].axis("off")

ax[1].imshow(img, cmap="gray")
ax[1].set_title("Original image")
ax[1].axis("off")

plt.show()
```

```
(145, 145)
(580, 580)
```



---

title: “Task 4”

Task 4

Data Handling and Preprocessing (10 Points)

a. You can focus for now on loading the T1-weighted images and the matching labels.

We used the `LoadImaged` transform from MONAI to load multiple modalities (`t1`, `t1ce`, `t2`, `flair`) and corresponding labels. The images are concatenated into a single tensor using the `ConcatItemsd` transform, and the labels are remapped using `RemapLabels` to ensure the labels are correctly handled.

b. Create a dataloader for the data using PyTorch's Dataloader (or Monai's Dataloader class).

The dataloader is created using the `get_dataloader` function. This function: - Reads the split files (e.g., `train.txt`, `validation.txt`, `test.txt`) containing dataset paths. - Applies a transformation pipeline (`get_transforms`) that preprocesses and augments the data. - Uses MONAI's `CacheDataset` for efficient loading and preprocessing. - Wraps the dataset in PyTorch's `DataLoader` to iterate over batches during training and validation.

c. Create suitable augmentations for the task to solve. Please note: If you apply transformations to the input data, you should think about if you need to apply any transformation to the label of the image as well.

Augmentations are applied in `get_transforms`: - `RandFlipd` flips the images and labels along different axes. - `RandSpatialCropd` crops the images and labels to the specified `roi_size`. The transformations account for labels by applying the same operations to both images and labels.

the code applies several augmentations in the `get_transforms` function: - **Spatial augmentations**: - `RandFlipd` flips the image and label along all three spatial axes with a probability of 0.5. - `RandSpatialCropd` crops both the images and labels to a predefined `roi_size`. - **Normalization**: - `NormalizeIntensityd` normalizes the image intensities channel-wise.

2. Data Splitting (5 Points)

a. Think about an appropriate split of the dataset into train, validation and test split. Briefly explain why it is good scientific practice to have separate Training, Validation and Test dataset?

Separating these datasets prevents overfitting and ensures that: - **Training dataset** is used for learning. - **Validation dataset** is used for tuning hyperparameters and evaluating performance during training. - **Test dataset** is used for final unbiased evaluation of the model's performance.

b. Please define the patient ids for each split in a separate file, like test.txt or validation.yml

Patient IDs are loaded from JSON files (`train.txt`, `validation.txt`, `test.txt`) in the `split_dir`. This ensures reproducibility and consistency in data splits.

3. Model Selection and Training (20 Points)

a. Implement a training pipeline to train the U-Net model of MONAI (or your own implementation if you want). Make use of the dataloader you created.

The training pipeline is implemented in the `train()` function: - The MONAI `UNet` is initialized with: - `spatial_dims=3` (to handle 3D data). - The number of input channels matches the modalities (`t1`, `t1ce`, `t2`, `flair`), and the number of output channels corresponds to the number of segmentation classes. - Customizable encoder-decoder channels and strides. - The pipeline uses the dataloaders (`train_loader` and `val_loader`) generated from the `get_dataloaders()` function, which provides training and validation datasets with proper transformations and augmentations. - The optimizer is `Adam`, and the loss function is defined as the `DiceLoss`.

b. At the end of each epoch, you should run a validation of the models performance.

Yes, the pipeline includes a validation step at the end of each epoch: - The validation data is processed through the model using `sliding_window_inference` to handle 3D volumes. - The Dice metric is calculated using the MONAI `DiceMetric` class to assess the overlap between predictions and ground truth. - Validation metrics, including the Dice score for each class and the mean Dice score, are logged for monitoring performance.

c. As we do not want to waste unnecessary energie and time, please implement an early stopping condition, that stops the training when the performance of the model does not improve for a few epochs on the validation set.

Yes, early stopping is implemented: - The `train()` function monitors the mean Dice score on the validation set. - If the Dice score does not improve for a defined number of epochs (default: `10`, controlled by `config.early_stop_limit`), training is stopped early to save time and computational resources.

d. Which loss function do you think suitable to train your model for this segmentation task? The `DiceLoss` is used for training, as it is well-suited for segmentation tasks. This loss directly optimizes the Dice score, which is a common metric for medical image segmentation. It ensures accurate overlap between the predicted and ground truth masks.

e. Please save the weights of the model that you trained. You can either save the weights after each epoch seperatly or you can override the previously saved model with the new best performing one after each validation.

The model weights are saved during training: - After each epoch, if the validation mean Dice score improves, the current model's weights are saved as `best_model.pth` in the specified `config.save_path` directory. - Only the best-performing model is saved, ensuring the most optimal model is retained without using unnecessary storage for suboptimal models.

4. Testing and Performance Evaluation (15 Points)

a. Think about a suitable metric to evaluate the performance of your model on the test set. You can also use more than one metric to capture different aspects of the models performance. The suitable metric for evaluating the model's performance on the test set is the **Dice score**, as it measures the overlap between predicted segmentation masks and ground truth labels. This metric is commonly used in medical image segmentation tasks.

b. Briefly explain, which model should you evaluate? You get a trained model after each epoch? Should you evaluate all of them and pick the best performing one?

- The best-performing model is saved during training (`best_model.pth`) based on the highest mean Dice score on the validation set.
- This saved model should be evaluated on the test set, as it represents the model with the best generalization during training and validation.
- Evaluating all models from every epoch would be unnecessary and computationally expensive, as the best model has already been identified using the validation set.

c. Code a test pipeline to evaluate your model's performance.

5. Extension to New Modalities (5 Bonus Points)

a. Extent the model to include multiple modalities of your choice as input (T2, T1ce, …). Justify your choice. The modalities are combined into a single tensor using `ConcatItemsd` in the `get_transforms` function.

- Different MRI modalities capture complementary information about the tumor:
  - **T1** and **T1ce** highlight the tumor core and enhancing regions.
  - **T2** and **FLAIR** are useful for visualizing edema and non-enhancing tumor areas.
- Combining modalities allows the model to leverage richer information, leading to improved segmentation performance and better differentiation between tumor sub-regions.

b. Train the model with the same data split and asses the performance of the model using your test pipeline. The training pipeline is designed to support multiple modalities as input:

The dataloader loads all modalities (t1, t1ce, t2, flair), and the concatenated tensor is passed to the UNet model for training. Using the same data split, the performance of the model can be assessed by running the `test_model()` function on the test set after training. The test pipeline evaluates the model's performance using metrics such as the Dice score.

c. Why and how does the performance change now that you use multiple modalities compared to using a single T1-weighted image as input?

Each modality highlights different tumor characteristics. For example: - **T1ce** captures enhancing regions. - **T2** captures edema and fluid buildup. - **FLAIR** is effective for visualizing non-enhancing tumor areas. - Using multiple modalities provides the model with diverse and complementary information, enabling it to make more accurate segmentation predictions.

Arora, Siddharth, Jayadev Acharya, Amit Verma, and Prasanta K Panigrahi. 2008. "Multilevel Thresholding for Image Segmentation Through a Fast Statistical Recursive Algorithm." *Pattern Recognition Letters* 29 (2): 119–25.

Liao, Ping-Sung, Tse-Sheng Chen, Pau-Choo Chung, et al. 2001. "A Fast Algorithm for Multilevel Thresholding." *J. Inf. Sci. Eng.* 17 (5): 713–27.