ML in DH (Assignment 02)

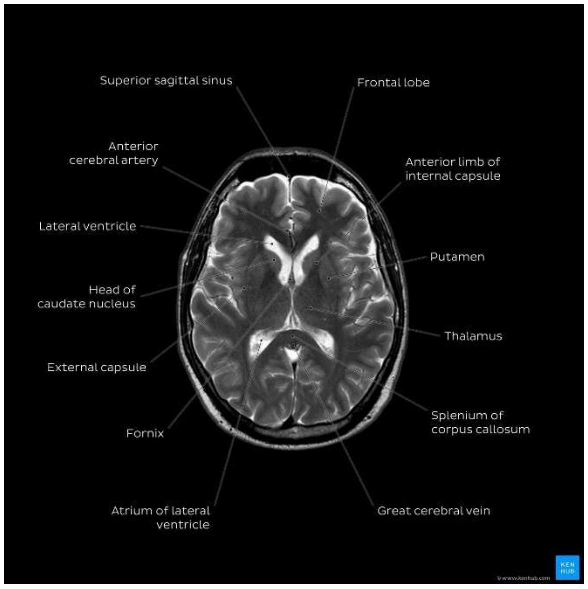
Overview:

• Brain Tumor segmentation on MRI (Magnetic Imaging Resonance) images using U-Net Architecture (CNN and Deep Learning):

Introduction and Problem Statement:

Tumors are groups of cells which form abnormal tissue or growths within the human anatomy. Tumors can either be malignant where the growth is cancerous and will invade surrounding cells, or benign where the suspected growth is not cancerous.

Brain tumors are such abnormal growths found within the human cranium. Given the complex and sensitive nature of the brain, a non-invasive technology, i.e. Magnetic Resonance Imaging (MRI), is the most popular pick for brain tumor diagnosis. These images are three-dimensional scans of a patient's brain and can be visualized on either of its three respecting image planes (Coronal, Sagittal and Transversal), each perspective plane displays its information regarding a potential abnormal growth within the cranium.



Brain tumor segmentation *aims to autonomously and accurately identify the size and location of a brain tumor from MRI scans. While traditional machine learning techniques require hand crafted features to perform well, most of the current research is focused on using deep learning networks to segment a region of interest (ROI) from an input image.*

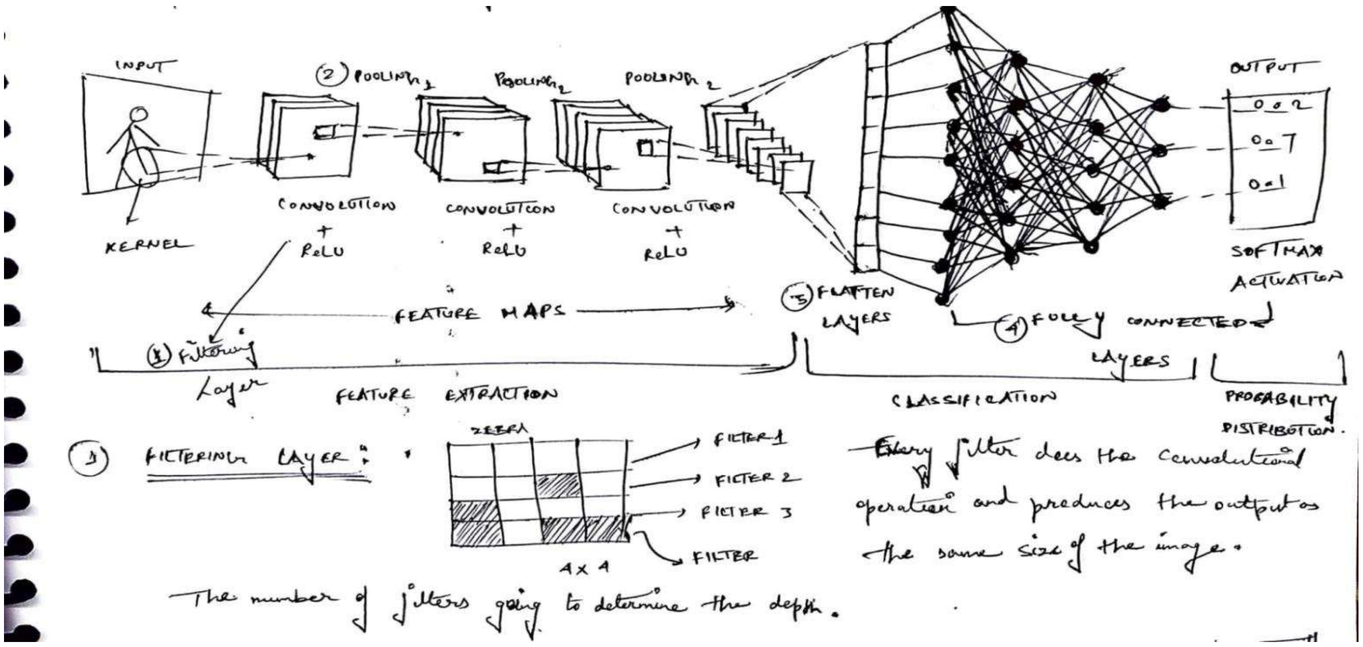
Q) What is a Convolutional Neural Network (CNN) ?

-> CNNs are powerful Deep leaning models designed for processing images and spatial data. They use specialized layers to detect patterns, edges and objects, making them perfect for image recognition, object detection, Medical image analysis and many more.

• Key Components of CNN :

i) ***Convolutional layer :*** *Extracts features using filters (kernels) to detect edges, textures etc.*
ii) ***Pooling Layer :*** *Reduces dimensionality while keeping important features and makes predictions (Max-Pooling is common).*
iii) ***Fully Connected layer :*** *Flattens extracted features and makes predictions.*
iv) ***Activation Functions :*** *Uses ReLU (Rectifier Linear Unit) to introduce Non-Linearity.*
v) *SoftMax / Sigmoid : Helps to classify images into categories.*

• CNN Architecture :



Every filter does the convolutional operation and produces the output as the same size of the image.

The number of filters going to determine the depth.

Q) Why CNNs are so powerful ?

-> ***CNNs*** *have revolutionized AI-Powered Vision :*

i) It automatically learn spatial hierarchies of features

ii) It reduces computational cost with parameter sharing.

iii) Works well with computer vision tasks.

• *Understanding U-Net Architecture :*

U-Net is a convolutional neural network designed specifically for image segmentation tasks. Its architecture consists of two main parts: the encoder (contracting path) and the decoder (expanding path).

*The **U-Net architecture** is based on an autoencoder network where the network will copy its inputs to its outputs. An autoencoder network functions by compressing the input image into a latent-space representation which is simply a compressed representation of the images indicating which data points are closest together.*

*The compressed data is later reconstructed to produce an output. An autoencoder network contains two paths, an **encoder** and **a decoder**. The encoder compresses the data into a latent space representation while the decoder is used for the reconstruction of the input data from its latent-space representation.*

***U-Net uses a convolutional autoencoder architecture** where the convolutional layers are used to encode and decode the input images.*

•Link to the code and Dataset : https://drive.google.com/file/d/1AyI8TlIR_dZ2Cl-ud3FOm4gwnY6pt0dk/view?usp=drive_link

Implemented Code : (Environment : Google Colab)

i) Importing Libraries in Python :

There are several libraries are used in implementation of python architecture such as NumPy, pandas, sci-kit learn, StandardScaler etc which is serving as a domain for finding the accuracy and prediction results to our problem statement.
*Additionally, APIs for Deep learning like **TensorFlow, keras and cv2** are used as they simplifies the process of building the Neural Networks.*

ii) Loading the Dataset :

We have to provide the link to the dataset by uploading the zipfile as the dataset contains approx. 3000 of training and testing "Grayscale" images in the following path for representation of the dataset in the code.

```
➔ from zipfile import ZipFile
file_name="/content/archive (2).zip"
with ZipFile(file_name,'r') as zip:
zip.extractall()
```

```
print('Done')
```

iii)
```
data=[], paths=[], result=[], masks=[], images=[], labels=[], for r,d,f in
os.walk(r'/content/brain_tumor_dataset/yes'):, for file in f:,  if '.jpg' in
file:,  paths.append(os.path.join(r,file)),  for path in paths:,
img=Image.open(path),  img=img.resize((128,128),  img=np.array(img),  if(img.shape==(128,128,3)):
,  data.append(np.array(img),  result.append(encoder.transform([[0]]))
```

➔ □ *data: to store image arrays (pixel values).*
➔ □ *paths: to store file paths.*
➔ □ *result: likely to store encoded labels.*
➔ □ *masks, images, labels: seem unused in this snippet.*

➔ *Opens each image.*
➔ *Resizes it to `128x128` pixels.*
➔ *Converts it to a NumPy array.*
➔ ***`encoder`** is likely a **label encoder**, such as `sklearn.preprocessing.OneHotEncoder` or `LabelEncoder`.*
➔ ***`transform([[0]])`** is converting the label `0` (which here probably means **'yes' tumor present**) into its encoded form.*
➔ *`[[0]]` is wrapped in double brackets because **`OneHotEncoder`** expects 2D input.*
➔ *These lines is **assigning the label 0 (for tumor-present class 'yes') to the image** in a machine-learning-friendly format.*

iv)
```
def load_image(file_path, target_size=(128, 128)):
  image = tf.keras.preprocessing.image.load_img(file_path, color_mode="grayscale", target_size=target_size)

image = tf.keras.preprocessing.image.img_to_array(image)

image = image / 255.0  # Normalize to [0, 1],    return image
```

➔ □ *Loads the image from file_path.*
➔ □ *Color_mode="grayscale": converts the image to a single channel (black and white).*
➔ □ *Target_size=target_size: resizes the image to 128×128 (or any size you pass)*
➔ *Since it's grayscale, the shape becomes `(128, 128, 1)`.*
➔ *Each pixel has a value in the range **[0, 255]**, and there's one channel.*
➔ *Normalizes pixel values from the standard 0–255 range to 0–1.*

0 -> Fully Black
255 -> Fully White

v)
```
%matplotlib inline
```

```
import matplotlib.pyplot as plt
plt.figure(figsize=(48,48))
for i in range(8):
plt.subplot(1,8,i+1)
plt.imshow(data[i],cmap="gray")
plt.axis("off")
plt.show()
```

- ➔ *Imports the matplotlib.pyplot module (usually used for plotting graphs and images) as plt.*
- ➔ ***figsize**=(48,48) makes a **very large figure** — this is likely overkill for showing just 8 images.*
- ➔ ***cmap="gray"** ensures it is shown in grayscale, even if it's a single-channel image.*
- ➔ *calling **plt.show()** inside the loop causes each image to be plotted on the system*

vi)
```
x_train,x_test,y_train,y_test = train_test_split(data, result, test_size=0.2,
shuffle=True, random_state=0)
print(f'Number of images in training data: {len(x_train)}')
print(f'Number of images in testing data: {len(x_test)}')
```

- ➔ *splits your dataset into **training and testing sets** using train_test_split*
- ➔ ***Input Parameters :** This is the label set (encoded labels corresponding to each image).*
- ➔ ***test_size=0.2**:*
  - *20% of the data will go into the **test set**.*
  - *The remaining 80% will be used for **training**.*
- ➔ ***shuffle=True**:*
  - *Randomly shuffles the data **before splitting**.*

vii)
```
fig=plt.figure(figsize=(20,12))
fig=plt.figure(figsize=(20,12))
for i in range (15):
  ax=fig.add_subplot(2,10,i+1,xticks=[],yticks=[])
  ax.imshow(np.squeeze(x_train[i]),cmap='gray')
  ax.set_title(y_train[i])
```

- ➔ *This initializes a new `matplotlib` figure for plotting.*
- *`figsize=(20,12)` sets the figure size in inches (width x height).*

`ax.imshow(np.squeeze(x_train[i]), cmap='gray')`

- Displays the image using `imshow()`.

```
Viii) def unet(input_size=(128, 128, 1)):
    inputs = tf.keras.layers.Input(input_size)

    # Contracting path (Encoder)
    c1 = tf.keras.layers.Conv2D(64, (3, 3),
activation='relu',padding='same')(inputs)
 c1 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu',
padding='same')(c1)
    p1 = tf.keras.layers.MaxPooling2D((2, 2))(c1)

    c2 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu',
padding='same')(p1)
    c2 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu',
padding='same')(c2)
    p2 = tf.keras.layers.MaxPooling2D((2, 2))(c2)
    c3 = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same')(p2)
    c3 = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same')(c3)
    p3 = tf.keras.layers.MaxPooling2D((2, 2))(c3)
    c4 = tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same')(p3)
    c4 = tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same')(c4)
    p4 = tf.keras.layers.MaxPooling2D((2, 2))(c4)

    # Bottleneck
    c5 = tf.keras.layers.Conv2D(1024, (3, 3), activation='relu', padding='same')(p4)
    c5 = tf.keras.layers.Conv2D(1024, (3, 3), activation='relu', padding='same')(c5)
   # Expansive path (Decoder)
    u6 = tf.keras.layers.Conv2DTranspose(512, (2, 2), strides=(2, 2),
padding='same')(c5)
    u6 = tf.keras.layers.concatenate([u6, c4])
    c6 = tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same')(u6)
    c6 = tf.keras.layers.Conv2D(512, (3, 3), activation='relu', padding='same')(c6)

    u7 = tf.keras.layers.Conv2DTranspose(256, (2, 2), strides=(2, 2),
padding='same')(c6)
    u7 = tf.keras.layers.concatenate([u7, c3])
    c7 = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same')(u7)
    c7 = tf.keras.layers.Conv2D(256, (3, 3), activation='relu', padding='same')(c7)

    u8 = tf.keras.layers.Conv2DTranspose(128, (2, 2), strides=(2,
2)padding='same')(c7)
    u8 = tf.keras.layers.concatenate([u8, c2])
    c8 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same')(u8)
    c8 = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same')(c8)
    u9 = tf.keras.layers.Conv2DTranspose(64, (2, 2), strides=(2,
2),padding='same')(c8)
    u9 = tf.keras.layers.concatenate([u9, c1])
    c9 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same')(u9)
    c9 = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same')(c9)

    outputs = tf.keras.layers.Conv2D(1, (1, 1), activation='sigmoid')(c9)
    model = tf.keras.Model(inputs=[inputs], outputs=[outputs])

    return model
```

➔ ***Encoder:***
*The encoder path follows the typical architecture of a convolutional network. It consists of repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a **rectified linear unit (ReLU)** and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step, we double the number of feature channels. This downsampling path captures the context of the image, extracting high-level features while reducing spatial dimensions.*

- ***3x3 Convolutions:*** *These are used to capture fine details in the image by applying small filters, helping in extracting detailed spatial features.*
- ***ReLU Activation:*** *Introduces non-linearity into the model, allowing it to learn complex patterns and speeds up the training process.*
- ***2x2 Max Pooling:*** *Reduces the spatial dimensions of the feature maps, decreasing the computational load and capturing significant features. This pooling provides translation invariance, making the network less sensitive to small translations of the input image.*

➔ ***Bottleneck:***
The bottleneck layer is the bridge between the encoder and decoder parts of the network. It captures the most abstract features of the input image at the lowest spatial resolution.

- ***Explanation:***
*The bottleneck consists of two 3x3 convolutional layers with **ReLU** activation, capturing the most abstract and high-level features from the input image. This part of the network focuses on the essence of the features, disregarding the exact spatial locations.*

➔ ***Decoder:***
The decoder path is symmetrical to the encoder and is designed to upsample the feature maps back to the original image size. It consists of an upsampling of the feature map followed by a 2x2 convolution ("up-convolution") that halves the number of feature channels.

- ***i) Up sampling (Up-Convolution):*** *Increases the spatial dimensions of the feature maps back to the original size of the input image. It helps generate a high-resolution segmentation map by combining detailed features from the encoder with high-level features from the decoder.*

- ***ii) 3x3 Convolutions:*** *Applied after upsampling to refine the features and ensure high-resolution output.*
-

Output Layer:

The final layer is a 1x1 convolution to map each 64-component feature vector to the desired number of classes (in this case, one for binary segmentation).

- ***1x1 Convolution:*** *Reduces the number of feature maps to the desired number of output channels, producing the final segmentation map with each pixel value representing the probability of the corresponding pixel belonging to the tumor class.*

viii) ***model = unet(input_size=(128, 128, 1)) model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4), loss='binary_crossentropy', metrics=['accuracy', tf.keras.metrics.MeanIoU(num_classes=2)]) model.summary()***

- ➔ *input_size=(128, 128, 1):*
- ➔ *The model expects input images of size **128x128 pixels**.*
- ➔ *1 channel means **grayscale** images.*
- ➔ ***optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4)***

- *Uses the **Adam optimizer (Adaptive Moment Estimation)** , a popular gradient-based optimizer.*
- *learning_rate=1e-4 (or 0.0001) sets the step size for weight updates — fairly typical for U-Net training.*

  ➔ ***loss='binary_crossentropy'***

- *It computes the loss between predicted and true pixel-wise class (0 or 1).*
- ***model.summary()***

  *Displays a **layer-by-layer breakdown** of the model architecture:*
- *Layer names*
- *Output shapes*
- *Number of trainable parameters*

ix) ***Model Training :***

*This function **trains your model** over several **epochs**, using your training and validation data.*

• x_train, y_train

- *These are your **training images** and their **corresponding labels/masks** (usually 0 = background, 1 = tumor).*
- *The model uses this to **learn**.*

  •Hyperparameter Tuning
- ***epochs=30***
- *The training process runs for **30 full passes** over the training dataset.*
- *More epochs = more learning opportunities, but risk of **overfitting** if too many.*

`batch_size = 40;` *The model processes data in **batches of 40 images at a time**.*

x)

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Test', 'Validation'], loc='upper right')
plt.show()
```

- ➔ *This plots the **training loss** over each epoch*
- ➔ *history is an object returned by model.fit(...) in Keras.*
- ➔ *'loss' corresponds to the **loss on the training data**. plt.ylabel('Loss')*
- ➔ *Labels the y-axis as "Loss" (indicating the loss value).*
- ➔ ***plt.show()** : Displays the plot.*

*Xi) **def names(number): if number==0: return 'Its a Tumor' else: return 'No, Its not a tumor'***

Input (number)	***Output (Meaning)***
0	*'Its a Tumor'*
1	*'No, Its not a tumor'*

xi) from matplotlib.pyplot import imshow img = Image.open(r"/content/brain_tumor_dataset/no/19 no.jpg") x = np.array(img.resize((128,128))) x = x.reshape(1,128,128,3) res = model.predict_on_batch(x) classification = np.where(res == np.amax(res))[1][0] imshow(img) print(str(res[0][classification]*100) + '% Confidence This Is ' + names(classification))

- ➔ *� Uses PIL (Image) to open an image from the specified file path.*
- ➔ *� This image likely belongs to the 'no' (no tumor) category in your dataset.*
- ➔ ***x = np.array(img.resize((128,128)))***

- *Resizes the image to **128x128 pixels** to match the model's input size.*
- *Converts the resized image to a NumPy array so it can be used by the model.*

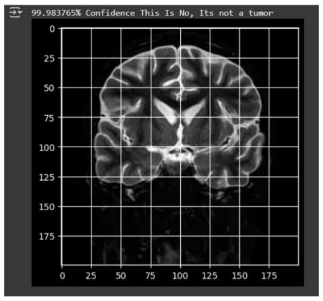
<u>res = model.predict_on_batch(x)</u>

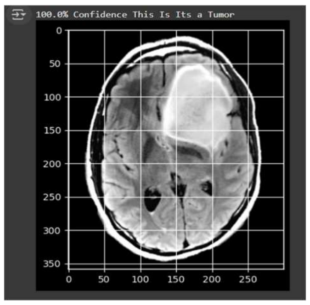
- *� Runs a prediction on the image using the model (for a batch of one image).*
- *� res will be a 2D array, like [[0.85, 0.15]] — meaning 85% tumor, 15% no tumor, or vice versa, depending on label order.*
- *classification will be 0 or 1, depending on which class had the highest score.*

- ➔ ***print(str(res[0][classification]*100) + '% Confidence This Is ' + names(classification))***

- *This gives the confidence to the model to predict whether the class contains Tumor or not.*

The accuracy is showing 99.983765 % for "It's not a tumor" and 100.00 % for "It's a tumor"


99.983765% Confidence This Is No, Its not a tumor


100.0% Confidence This Is Its a Tumor

Conclusion

In this study, we explored how U-Net, a well-established deep learning architecture for biomedical image segmentation, can be adapted and fine-tuned for the specific task of brain tumor segmentation. Our implementation of U-Net is lightweight yet effective, capable of delivering accurate segmentation results without relying heavily on aggressive data augmentation techniques. This makes it particularly suitable for practical applications. The proposed network holds potential for use in medical settings, serving as a valuable second-opinion tool for trained physicians when analyzing MRI images.