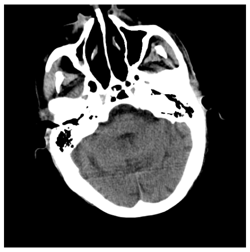
Tutorial on CNN for Brain Hemorrhage Classification in CT Images

The objective of this tutorial is to describe the Python code step by step to implement a Convolutional Neural Network (CNN) for brain hemorrhage classification.

Overview of CT Images of the Human Brain:
Computed Tomography (CT) of the head plays a crucial role in the rapid assessment of **brain hemorrhage**. It uses specialized X-ray imaging to detect and evaluate bleeding within the brain tissue or surrounding areas, such as the subarachnoid space, ventricles, or subdural/epidural regions. CT imaging[1] is essential in diagnosing patients presenting with symptoms like sudden severe headache, loss of consciousness, neurological deficits, or trauma-induced brain injury.



How does it work?
CT imaging of the brain uses X-rays taken from multiple angles around the head. These X-ray signals are processed by a computer to create detailed cross-sectional images of the brain. X-rays absorb the areas with different tissue densities, like blood, bone, and brain differently, allowing the CT scan to clearly reveal abnormalities like hemorrhages, tumors, or swelling [1,2].

What It Shows?
A brain CT scan clearly displays different structures such as the skull, brain tissue, cerebrospinal fluid spaces, and blood vessels. It can detect abnormalities like hemorrhages, tumors, swelling (edema), fractures, and signs of stroke. In cases of brain hemorrhage, CT shows areas of bleeding as bright (hyperdense) regions compared to the normal brain tissue.

1. Slice Thickness: The Zoom Level of the Scan

A CT scan is a stack of digital "slices." Thin slices (e.g., 0.625mm) are like high-resolution, zoomed-in images, which are excellent for spotting tiny, subtle problems like a very fine skull fracture or a small bleed along the surface of the brain. On the other hand, thick slices (e.g., 5mm) are like a zoomed-out overview. They provide a clearer, less "noisy" picture of the overall brain structure, making it easier to see larger hemorrhages quickly. Both thin-slices and thick-slice are needed respectively to have finer details and clear overview.

2. Reconstruction Kernel: The Focus of the Image

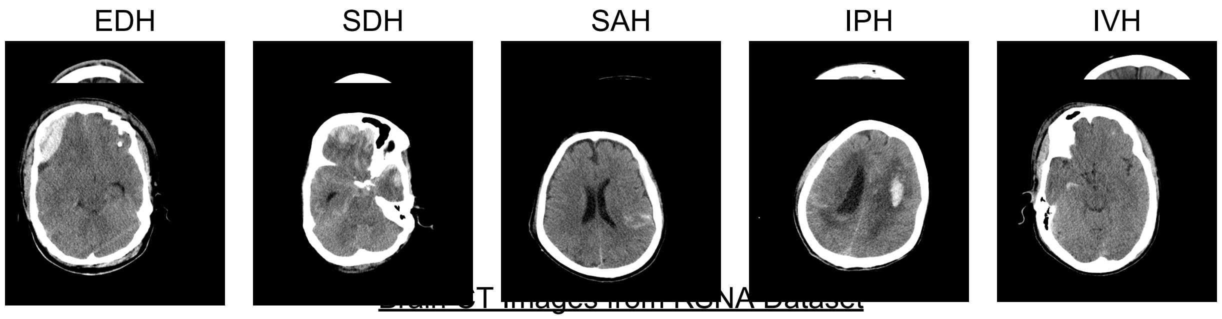
Soft Tissue / Brain Kernel: This setting smoothen the CT image to make it easy to see differences in the brain's soft tissues, hence useful for identifying hemorrhage within the brain.

[1] https://europepmc.org/article/nbk/nbk567796#free-full-text

Bone Kernel: This setting enhances edges and makes the image sharper. It's terrible for looking at the brain (it makes it look grainy), but it's perfect for seeing fine-line fractures in the dense skull bone.

3. Contrast: Lighting Up the Blood Vessels

Non-Contrast CT is the standard emergency scan, fast and useful for finding fresh blood in the brain. For **Contrast-Enhanced CT (CECT)**, a special iodine-based dye is injected into the bloodstream. This dye lights up on the CT scan, making blood vessels appear bright white. This is incredibly useful for finding the source of a bleed, like an aneurysm (a weak bulge in a blood vessel) or a tumour with an abnormal blood supply.



Brain CT Images from RSNA Dataset

Brain CT Images from GMCB Dataset

Advantages and Limitations:

CT scans help doctors see the brain quickly and clearly, which is very useful in emergencies like head injuries or strokes. They are available in most hospitals and show details of bones, such as skull fractures, very well. However, CT scans also have some limits. CT scans use radiation, which poses a small health risk, and some individuals may have reactions to the contrast dye. Also, CT is not as effective as MRI for showing soft tissues like the brainstem. Reading and understanding CT images are properly taken with skill and time, so experts are needed to make an accurate diagnosis.

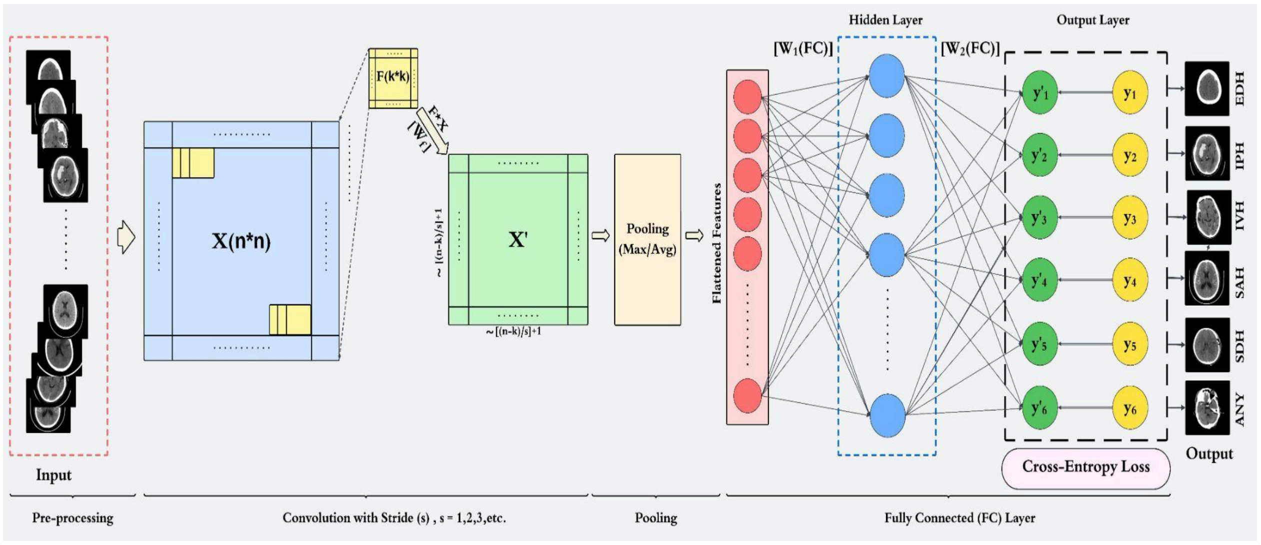
Overview of Brain Hemorrhage Classification using CT Images:

Our goal is to detect and classify the type of hemorrhage from the Brain CT scans, for this we prepare a dataset or a csv file where each entry consists of an image path pointing to a CT scan and 6 binary columns. Five of these columns represent the following types of hemorrhages:
i) Epidural (EPH) ii) Subdural (SDH) iii) Subarachnoid (SAH) iv) Intraparenchymal (IPH) and v) Intraventricular (IVH) Hemorrhage. The sixth class is called ANY Class [1]. The first five places in the output column will be marked as 1, if a particular type from five different hemorrhages is present in the CT scan in a fixed order and 0 if it is not. The sixth column is 'ANY', which is set to 1, if any of the hemorrhages are present, and 0 if all the other five hemorrhage columns are 0. The first set of images in the figure above are from *Radiological Society of North America (**RSNA**[2]) Datase*t [2], and the second set of images are from an Indian dataset from *Gandhi Medical College Bhopal (**GMCB**).*

Background of CNN:

[2] https://www.kaggle.com/competitions/rsna-intracranial-hemorrhage-detection

Convolutional Neural Networks are deep learning models designed to process data with a grid-like topology such as images [3,4,5]. Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed for processing grid-like data, such as images [4]. The foundational concepts of CNNs were inspired by the 1960's Nobel Prize-winning research of neurobiologists Hubel and Wiesel. Their experiments showed that neurons in a cat's visual cortex are organized in a hierarchy. Some neurons fired only in response to simple patterns like vertical lines, while others, receiving signals from these simple neurons, fired for more complex shapes. This biological principle was computationally modeled in the 1980s and 90s, most notably with Yann LeCun's LeNet-5.



Architecture of Classical CNN for Brain Hemorrhage Classification in CT Scans

The core innovation of CNNs lies in their specialized layers as depicted in the figure above. Convolutional layers apply filters (or kernels) that slide across an input image, detecting specific features like edges, textures, and shapes [3]. This process creates "feature maps" that highlight where these patterns occur. Following this, pooling layers often downsample these maps, reducing the data's size, which makes the network more efficient and robust to small shifts in an object's position. This structure allows CNNs to build a hierarchy of features. Early layers learn simple patterns, while deeper layers combine these to recognize more complex structures—from edges to shapes, and ultimately to entire objects like faces or cars.

Dataset Description and Preparation:

For training our CNN model we would first need to transform our dataset. We have a dataset where each folder represents a patient, and inside each folder are different types of CT scans (e.g., Plain, Contrast, Thin Slice).

Each scan type is stored in a subfolder and contains multiple DICOM slices and our csv file contains the hemorrhage label for each scan. We need slice-level labels for our model so our first step would be to create another csv file to assign the scan level labels to all the DICOM slices within that scan.

Step-By-Step Implementation of CNN for Brain Hemorrhage (ICH) Classification

1. **Importing Libraries**

```
import torch
import torch.nn as nn
import torch.nn.functional as func
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import numpy as np
import pandas as pd
import torchvision
import torchvision.transforms as transforms
import pydicom
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score,
recall_score, confusion_matrix
from PIL import Image
```

At the core, **torch** provides the fundamental tools for constructing neural networks, managing tensors, and optimizing model training — it's the backbone of any PyTorch deep learning workflow [6]. Complementing this, **torch.utils.data.Dataset**[3] and DataLoader allow us to create custom datasets and load them efficiently in batches, making it easier to train models on large volumes of image data without exhausting system memory.

pydicom[4] is a python library that is used when dealing with medical imaging; it enables us to read and manipulate DICOM files — the standard format for storing CT, MRI, and other medical scan data.

sklearn.model_selection[5] ensures the model generalizes well by splitting data thoughtfully between training, and testing sets. It offers tools like train_test_split and cross-validation to help structure experiments.

Lastly, **torchvision.transforms**[6] offers a variety of image preprocessing and augmentation functions to prepare images for model input.

2. **Loading and Splitting Metadata**

```
if __name__ == "__main__":
    train_csv = '/content/train_sample.csv'
    test_csv = '/content/test_sample.csv'

    train_df = pd.read_csv(train_csv)
    test_df = pd.read_csv(test_csv)
```

[3] https://docs.pytorch.org/tutorials/beginner/basics/data_tutorial.html
[4] https://pydicom.github.io/
[5] https://scikit-learn.org/stable/api/sklearn.model_selection.html
[6] https://docs.pytorch.org/vision/main/transforms.html

We begin by using the `read_csv()` function in the pandas library that reads our metadata csv file from the file path and loads it into a pandas DataFrame.

```
label_cols = ['any', 'intraparenchymal', 'intraventricular',
'subdural', 'subarachnoid', 'epidural']

X_train = train_df['filename']
y_train = train_df[label_cols]

X_test = test_df['filename']
y_test = test_df[label_cols]
```

`label_cols` is a list of column names representing the target labels for classification.
`X_train/test = df['filename']:` Extracts the 'filename' column, containing the input image file paths, into a Pandas Series X.
`y_train/test = df[label_cols]:` Extracts the specified label_cols into a Pandas DataFrame y, containing the output labels.

```
train_data_directory = '/content/train_images' # Update this to your
actual path

test_data_directory = '/content/test_images' # Update this to your
actual path
```

Here we set the path of the directory which contains the training and test data. The path is written within *'path'*, i.e. single quotes to specify it is a string showing where to look for the respective images.

3. Defining Transformation Pipeline

Next the transformations that are to be applied to the images are defined:

- `transforms.Resize` resizes image to 512x512 pixels regardless of its original size.
- `transforms.ToTensor()` method converts datatype to Tensor to scale the pixel values between 0 and 1 and reorders the dimensions to channel, height and width.
- `transforms.Normalize()` method normalizes the image by subtracting the mean and dividing by the standard deviation. This ensures that pixel values are centered around 0 and have roughly unit variance, which helps the model to converge faster and perform better.

```
IMG_SIZE = 512
    transform = transforms.Compose([
        transforms.Resize((IMG_SIZE, IMG_SIZE)),
        transforms.ToTensor(),
        transforms.Normalize((0.380),(0.236)), ])
```

4. **Loading CT Scans and applying transformations**

```
class CTdataset(Dataset):
    def __init__(self, data_df, transform, data_dir):
        self.data = data_df.reset_index(drop=True)
        self.transform = transform
        self.data_dir = data_dir
        self.labels = [
            'any',
            'intraparenchymal',
            'intraventricular',
            'subdural',
            'subarachnoid',
            'Epidural'         ]
```

A custom class `CTdataset(Dataset)` is made which inherits from the `torch.utils.data.Dataset` base class. Inheriting from `Dataset` is a standard practice for creating custom datasets, enabling compatibility with `torch.utils.data.DataLoader` for efficient data loading during model training.
`def __init__(self, data_df, transform)` is the constructor which expects a dataframe parameter `data_df` and a transformation pipeline `transform.` Inside it some variables are designed:

- `self.data = data_df.reset_index(drop = True):` This method resets the DataFrame's index. The `drop=True` argument prevents the old index from being inserted as a new column in the DataFrame. It ensures that the DataFrame has a default integer index (0, 1, 2, ...), making indexing by integer position within the `__getitem__` method easy, regardless of the original index of the input data_df.

- `self.transform = transform` This line assigns the `transform` object passed to the constructor to an instance variable `self.transform.`

- The fourth line initializes an instance variable `self.labels` as a Python list of strings. Each string represents the name of a specific intracranial hemorrhage subtype or a general "any" hemorrhage classification.

```
def __len__(self):
        return len(self.data)
```

The `__len__()` method returns total number of samples in the given dataset.

```
   def __getitem__(self, index):
        filename = self.data.loc[index, 'filename']
        img_path = f"{self.data_dir}/{filename}"

        dcm = pydicom.dcmread(img_path)
        img = dcm.pixel_array.astype(np.float32)

        img_min, img_max = img.min(), img.max()
        img = (img - img_min) / (img_max - img_min + 1e-5)
        img = (img * 255).astype(np.uint8)

        img = Image.fromarray(img)
        image_tensor = self.transform(img)

                                                     label_series        =
self.data.iloc[index][self.labels].values.astype(np.float32)
        label_tensor = torch.tensor(label_series)

        return image_tensor, label_tensor
```

This code defines a __getitem__ method within a class.

- `img_path = self.data.loc[index, 'filename']`: This line retrieves the file path of an image at the specified index from the slice_path column of the data attribute of the class instance.

- `dcm = pydicom.dcmread(img_path)`: This line reads the DICOM file located at img_path using the pydicom library and stores the DICOM object in the variable dcm.

- `img = dcm.pixel_array.astype(np.float32)`: This line extracts the pixel array data from the DICOM object dcm and converts it to a NumPy array of type float32, storing it in the variable img.

- `img -= img.min()`: Subtracts the minimum pixel value from all pixels in the image. This shifts the pixel range so that the lowest value becomes 0.
- `img /= (img.max() + 1e-5)`: Normalizes the image so that pixel values are between 0 and 1. We divide by (max + 1e-5) where 1e-5 = 0.00001 to avoid division by zero (numerical stability).

- `img = (img * 255).astype(np.uint8)`: Scales the normalized image [0, 1] back to pixel values [0, 255]. , it also converts the image to 8-bit unsigned integers (a standard for image formats).

Illustration with Synthetic a Data Matrix:

$$\begin{pmatrix} 10 & 50 & 100 \\ 20 & 70 & 120 \\ 30 & 90 & 150 \end{pmatrix}$$

After subtracting the minimum value from all the elements of the matrix

$$\begin{pmatrix} 10-10 & 50-10 & 100-10 \\ 20-10 & 70-10 & 120-10 \\ 30-10 & 90-10 & 150-10 \end{pmatrix} = \begin{pmatrix} 0 & 40 & 90 \\ 10 & 60 & 110 \\ 20 & 80 & 140 \end{pmatrix}$$

After dividing all the elements by the maximum value:

$$\begin{pmatrix} 0/140 & 40/140 & 90/140 \\ 10/140 & 60/140 & 110/140 \\ 20/140 & 80/140 & 140/140 \end{pmatrix} \approx \begin{pmatrix} 0.00 & 0.29 & 0.64 \\ 0.07 & 0.43 & 0.79 \\ 0.14 & 0.57 & 1.00 \end{pmatrix}$$

After scaling it back to the [0,255] range:

$$\begin{pmatrix} 0.00 \times 255 & 0.29 \times 255 & 0.64 \times 255 \\ 0.07 \times 255 & 0.43 \times 255 & 0.79 \times 255 \\ 0.14 \times 255 & 0.57 \times 255 & 1.00 \times 255 \end{pmatrix} = \begin{pmatrix} 0 & 73.95 & 163.2 \\ 17.85 & 109.65 & 201.45 \\ 35.7 & 145.35 & 255 \end{pmatrix}$$

After casting it to np.uint8:

$$\begin{pmatrix} 0 & 73 & 163 \\ 17 & 109 & 201 \\ 35 & 145 & 255 \end{pmatrix}$$

```
img = Image.fromarray(img)
image_tensor = self.transform(img)
```

- `img = Image.fromarray(img)`: This line returns the NumPy array img back as an image object using PIL.Image.fromarray().
- `image_tensor = self.transform(img)`: This line applies the transformation pipeline to the image and converts it to a tensor.
- `label_series=self.data.iloc[index]self.labels].values.astype(np.float32)`: This line retrieves the labels corresponding to the image at the given index. It uses the instance variable self.labels (which was defined in the class constructor) to

select the correct label columns from the DataFrame, converts their values to a NumPy array of type float32, and stores them in label_series.

```
label_series =
self.data.iloc[index][self.labels].values.astype(np.float32)
label_tensor = torch.tensor(label_series)
return image_tensor, label_tensor
```

- `label_tensor=torch.tensor(label_series):` This line converts the NumPy array label_series to a PyTorch tensor label_tensor.
- Finally, the method returns a tuple containing image_tensor and label_tensor, representing the preprocessed image and its corresponding labels, respectively.

```
rain_dataset  =  CTdataset(data_df=train_df,  transform=transform,  data_dir  =
rain_data_directory)
est_dataset  =  CTdataset(data_df=test_df,  transform  =  transform,  data_dir  =
est_data_directory)
```

We create instances of our custom CTdataset class passing their respective data (train_df, test_df) along with the transformation pipeline. It is important to note that train_dataset and test_dataset are objects of the CTdataset class that hold references to the DataFrame and transform function.

5. Defining CNN for Brain Hemorrhage (ICH) Classification

```
class ConvNet(nn.Module):
    def __init__(self):
        super().__init__()
```

This code defines a new neural network class called ConvNet. It inherits from nn.Module, which is PyTorch's base class for all neural network modules. The __init__ method initializes the parent class, setting up the basic structure for the network.

The network architecture is as follows:

- The first convolutional layer (conv1) takes an input with 1 channel and applies 50 filters of size 5x5. The output feature map has a size of 508x508.

- A max pooling layer (pool) with a kernel size of 2 and a stride of 2 reduces the spatial dimensions by half, resulting in a feature map of size 254 x 254.
- The second convolutional layer (conv2) takes the 50 input channels from the previous layer and applies 120 filters of size 5 x 5. The output feature map is 250 x 250, which is then reduced to 125 x 125 after max pooling.

- The third convolutional layer (conv3) takes the 120 input channels and applies 100 filters of size 5 x 5. The output feature map is 121 x 121, which is then reduced to 60x60 after max pooling.

- The fourth convolutional layer (conv4) takes the 100 input channels and applies 72 filters of size 4 x 4. The output feature map is 57 x 57, which is then reduced to 28 x 28 after max pooling. The output is flattened to be fed into the fully connected layers.

- There are two fully connected layers (fc1 and fc2) with 72 x 28 x 28 input features and output sizes of 50 and 6, respectively.

Overall, this ConvNet architecture consists of multiple convolutional and pooling layers followed by fully connected layers for classification.

```
self.conv1 = nn.Conv2d(1, 50, kernel_size=5)
  self.conv2 = nn.Conv2d(50, 120, kernel_size=5)
  self.conv3 = nn.Conv2d(120, 100, kernel_size=5)  # Removed duplicate
  self.conv4 = nn.Conv2d(100, 72, kernel_size=4)
  self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
  self.fc1 = nn.Linear(72 * 28 * 28, 50)
  self.fc2 = nn.Linear(50, 6)
```

6. **The forward propagation method**

```
def forward(self, x):
      x = self.pool(torch.relu(self.conv1(x)))
      x = self.pool(torch.relu(self.conv2(x)))
      x = self.pool(torch.relu(self.conv3(x)))
      x = self.pool(torch.relu(self.conv4(x)))
      x = torch.flatten(x, 1)
      x = torch.relu(self.fc1(x))
      x = self.fc2(x)

      return x
```

This code defines the forward method that takes an input ‘x’, this input x is passed through a series of convolutional layers (‘self.conv1’, ‘self.conv2’, ‘self.conv3’, ‘self.conv4’) with ReLU activation functions applied after each convolution operation. After each convolution operation, the output is passed through a pooling layer (‘self.pool’).

The output from the last convolutional layer is then flattened into a 1-dimensional tensor using ‘torch.flatten’. The flattened tensor is passed through a fully connected layer (‘self.fc1’) with

ReLU activation function applied. Finally, the output is passed through another fully connected layer (‘self.fc2’) without any activation function applied. The final output is returned.

7. **Creating DataLoader**

```
 BATCH_SIZE = 16
 NUM_WORKERS = 2

 train_loader = DataLoader(
     train_dataset,
     batch_size=BATCH_SIZE,
     shuffle=True,
     num_workers=NUM_WORKERS
)

test_loader = DataLoader(
    test_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False,
    num_workers=NUM_WORKERS,
)
```

This code snippet is setting up data loaders for training and testing datasets using PyTorch's DataLoader class.

- BATCH_SIZE is set to 16, which means that the data will be loaded in batches of 16 samples during training and testing. It needs to be tuned based on the data set.

- NUM_WORKERS is set to 2, which specifies the number of subprocesses to use for data loading. This can speed up data loading as the data can be loaded in parallel using multiple workers.

For the training data loader and testing data loader:

1. train_loader is initialized with the ‘train_dataset’ and test_loader is initialized with ‘test_dataset’.
2. The batch_size parameter is set to BATCH_SIZE, so each batch will contain 16 samples.
3. shuffle is set to ‘True’, which means that the data will be shuffled at every epoch before creating batches.
4. num_workers is set to NUM_WORKERS, so 2 subprocesses will be used for data loading.

Overall, these data loaders will be used to load data in batches for training and testing machine learning models, with shuffling and parallel data loading for efficiency.

8. **Setup**

```
model = ConvNet()
loss_fn = nn.BCEWithLogitsLoss()
opt = optim.Adam(model.parameters(), lr=0.001)
device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device)
```

Here, the model variable is an instance of the convolutional network. For the loss function, we use BCEWithLogitsLoss, which combines the sigmoid function with BCELoss, making it ideal for predicting probabilities for multiple conditions in CT scans. Subsequently, the widely used **Adam optimizer** is used, which adjusts learning rates adaptively for each parameter, speeding up convergence and improving training stability. The learning rate is set to 0.001, which controls the step size during weight updates. We set the device where we want to run our model to **"cuda"**, which indicates that we want to **use a GPU** for computations. We move the model to the specified device by calling the to () method on the model and passing the device variable as an argument. This ensures that the model's computations will be performed on the GPU if available.

9. Training the Model

```
NUM_EPOCHS = 10

for epoch in range(NUM_EPOCHS):
    model.train()
    train_loss, train_preds, train_labels = 0, [], []
```

We initialize a variable NUM_EPOCHS with a value of 10. An epoch represents one complete pass through the entire training dataset. Inside the for loop, model.train () puts the model into training mode. Three variables train_loss, train_preds, and train_labels are initialized respectively to 0, and two empty lists. These variables will be used to keep track of the total training loss, predictions made by the model using the training data, and corresponding labels.

```
    for imgs, labels in train_loader:
        imgs = imgs.to(device)
        labels = labels.to(device)
```

A for loop is used to iterate over the train_loader, which is an object that provides batches of training data. In each iteration, the loop will extract a batch of images (imgs) and their corresponding labels (labels). Inside the loop, the images (imgs) and labels (labels) are being moved to the specified device (cuda) using the to() method.

```
        opt.zero_grad()
        logits = model(imgs)
        loss = loss_fn(logits, labels)
        loss.backward()
        opt.step()
```

- `opt.zero_grad()` is called to clear out any gradients from the previous iteration.

- `logits = model(imgs)` generates raw predictions (called logits) of the given images.
- `loss = loss_fn(logits, labels)` calculates the value of binary cross-entropy loss using the logits and ground truth labels
- `loss.backward()` computes the gradient of the loss with respect to all the learnable parameters in the model, following the chain rule of derivatives to propagate the gradients backwards through the computational graph.
- `opt.step()` updates the parameters of the model using the gradients computed in the previous step. Otimizer (opt) uses these gradients to adjust the weights of the model in the direction that minimizes the loss following an optimization technique (e.g., Adam).

```
        train_loss += loss.item() * imgs.size(0)
```

- `loss.item()` retrieves the scalar value of the loss function for a single training example in the batch. The item() method is used to extract the loss value as a Python number from a PyTorch tensor.
- `imgs.size(0)` retrieves the number of training examples in the batch. The size(0) method is used to get the size of the batch dimension of the input images.
- `loss.item() * imgs.size(0)` calculates the total loss for the entire batch by multiplying the loss value for a single example by the number of examples in the batch.
- `train_loss +=` adds total loss for the batch to the train_loss variable, where **+=** is used to incrementally add total loss over multiple batches during the training process.

In summary, this line of code is updating the train_loss variable by adding the product of the loss value for a single example and the number of examples in the batch.

- `torch.sigmoid(logits)` applies the sigmoid function to the input tensor logits.
- `(torch.sigmoid(logits) > 0.5)` compares each element of the tensor with 0.5 and returns a boolean tensor with the same shape as the input tensor.
- `.float()` converts the boolean tensor to a float tensor where True is represented as 1.0 and False is represented as 0.0.
- The resulting tensor preds contains the predictions where values greater than 0.5 are considered as 1 and values less than or equal to 0.5 are considered as 0.
- `train_preds.extend(preds.cpu().numpy())` extends the train_preds list with the predictions obtained from the current batch after moving it to the CPU and converting them to a NumPy array.
- `train_labels.extend(labels.cpu().numpy())` extends the train_labels list with the actual labels of the current batch after moving it to the CPU and converting them

to a NumPy array.

We then look at the *loss* and *accuracy* of the model during its training phase, while also saving the model for future testing, i.e., evaluating the model on an unseen set of test images.

```
            preds = (torch.sigmoid(logits) > 0.5).float()
            train_preds.extend(preds.cpu().numpy())
            train_labels.extend(labels.cpu().numpy())

        avg_train_loss = train_loss / len(train_dataset)
        train_preds = np.array(train_preds)
        train_labels = np.array(train_labels)
        train_acc = (train_preds == train_labels).mean()

        print(f"Epoch [{epoch+1}/{NUM_EPOCHS}] - Loss:
{avg_train_loss:.4f}, Accuracy: {train_acc:.4f}")

torch.save(model.state_dict(), 'hemorrhage_model.pth')
print("Model saved successfully!")
```

10. Evaluating the Model

```
    model.eval()
    val_loss, val_preds, val_labels = 0, [], []
```

- `model.eval()` sets the model to evaluation mode. In PyTorch, this is important because it disables dropout layers and batch normalization layers during inference to ensure consistent results.
- `val_loss, val_preds, val_labels = 0, [], []:` Three variables val_loss, val_preds, and val_labels are initialized, which will be used to store the validation loss, predicted outputs, and actual labels, respectively.

```
    with torch.no_grad():
        for imgs, labels in test_loader:
            imgs = imgs.to(device)
            labels = labels.to(device)
```

- `with torch.no_grad()` is a context manager provided by PyTorch to temporarily disable gradient calculation. This is useful during inference to reduce memory consumption and speed up computations since gradients are not needed for evaluation.

This is a loop that iterates over the test_loader, which contains batches of images that the model did not learn during training and their corresponding labels. The input images and their labels are moved to the specified device (CUDA) for faster computation. This is done to leverage the parallel processing capabilities of GPUs.

```
logits = model(imgs)
val_loss += loss_fn(logits, labels).item() * imgs.size(0)
preds = (torch.sigmoid(logits) > 0.5).float()
val_preds.extend(preds.cpu().numpy())
val_labels.extend(labels.cpu().numpy())
```

- `logits = model(imgs)` passes the input images through the model to get the logits, which are the raw outputs of the model before applying any activation function.
- `val_loss += loss_fn(logits, labels).item() * imgs.size(0):` calculates the loss between the logits and the actual labels, using the loss_fn function. The loss is then multiplied by the number of images in the batch (imgs.size(0)) and added to the val_loss variable, which is used to keep track of the total loss.
- `preds = (torch.sigmoid(logits) > 0.5).float()` stores binary class labels predicted by passing the logits through the sigmoid function to squash the values between [0,1] and then convert it into binary predictions based on a threshold (0.5 here).
- `val_preds.extend(preds.cpu().numpy())` converts binary predictions (preds) to a NumPy array on the CPU and then added to the val_preds list, which is used to store the predictions for the entire validation dataset.
- `val_labels.extend(labels.cpu().numpy())` converts the actual labels (labels) to a NumPy array on the CPU and then added to the val_labels list, which is used to store the actual labels for the entire validation dataset.

The code is creating NumPy arrays from the variables val_preds and val_labels and then flattening them into 1-dimensional arrays. The model calculates two primary metrics across the test dataset:
Test Loss computes binary cross-entropy loss averaged over all test instances to measure how well the model predictions are aligned with true labels across all six hemorrhage types together.

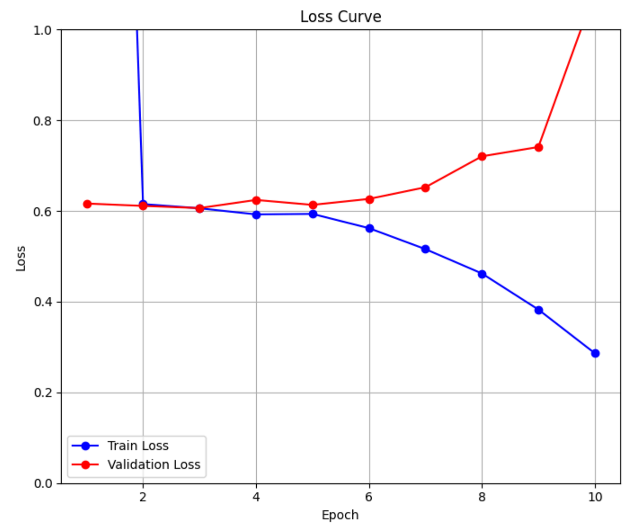
Test Accuracy is the proportion of correct predictions across all labels for all instances. This is calculated by comparing predicted labels (after applying a 0.5 threshold to the sigmoid outputs) with the true labels, then averaging across all predictions. This provides an overall measure of how accurately the model classifies hemorrhages at the slice level.

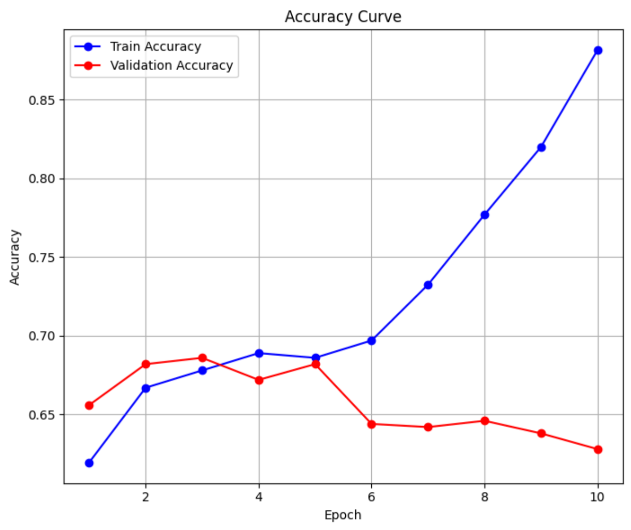
```
val_preds_np = np.array(val_preds)
val_labels_np = np.array(val_labels)
accuracy = (val_preds_np == val_labels_np).mean()

print(f"\n Test Loss: {val_loss / len(test_dataset):.4f}")
print(f"Test Accuracy: {accuracy:.4f}")
```

The evaluation processes images batch by batch, displaying progress for each image as it is evaluated. The final metrics provide a high-level view of the model's classification performance on unseen test data.

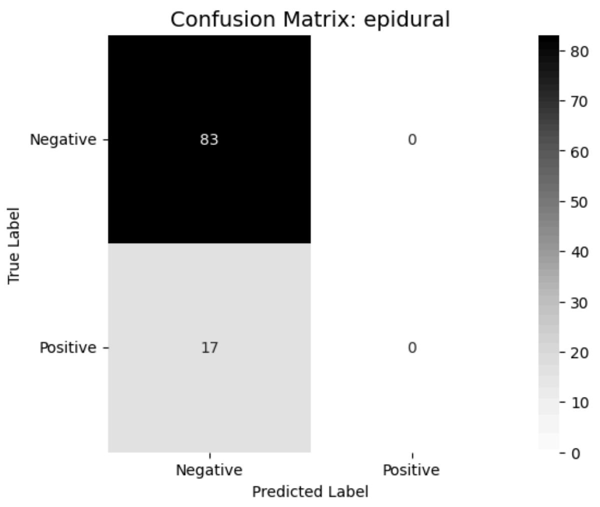
11. Evaluation Analysis

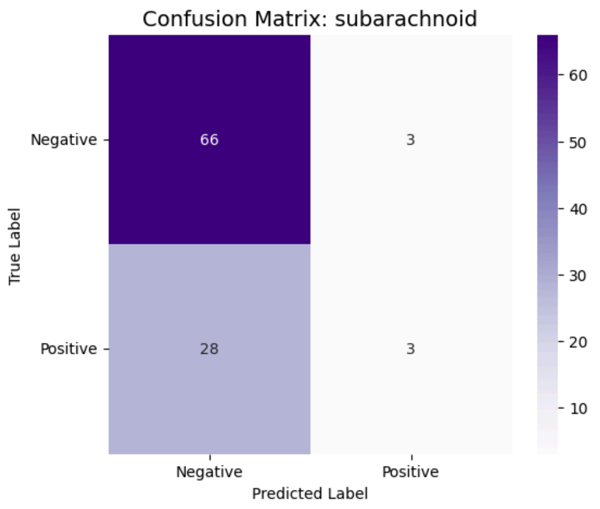


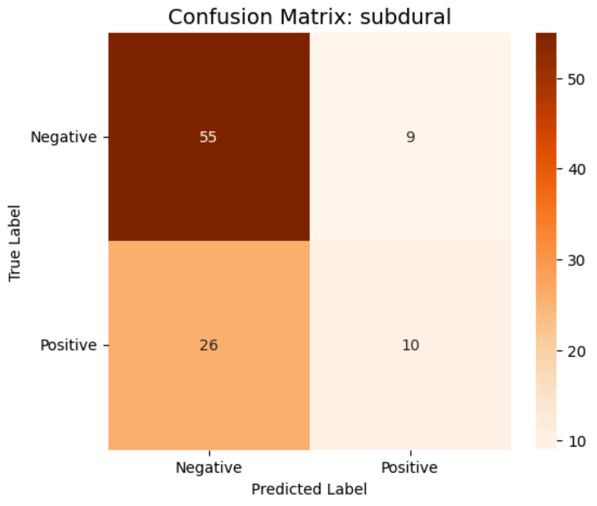


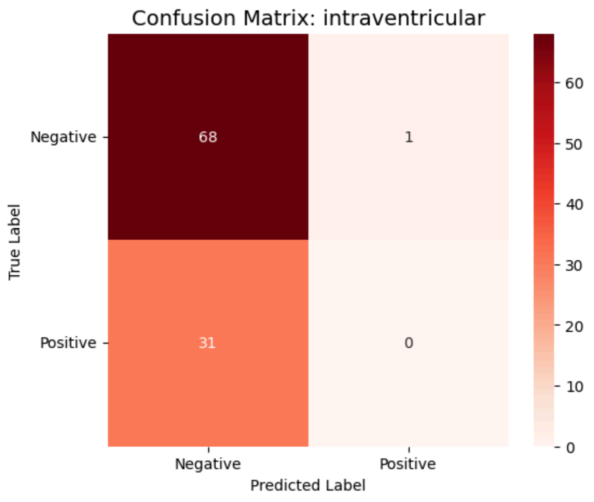
Loss Curve shows how the training and validation loss of the model changes over time, or in machine learning terms, ‘epochs’. We can see that, till epoch 3, the validation loss is the lowest, and then it starts increasing. In principle, the lower the loss, the better the model.
Accuracy Curve is used to see how well the model performs in predicting the correct classes. We can see that the training accuracy increases after epoch 6, whereas the validation accuracy decreases, providing a perfect example of model overfitting.

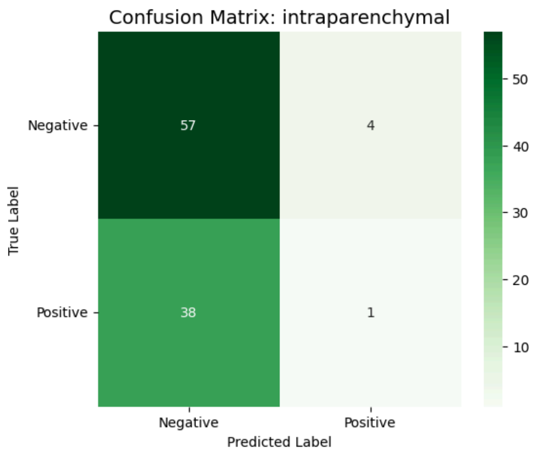
Confusion Matrices are often used to check the number of correctly identified or wrongly identified cases by a model during predictions. It consists of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN).
With these values, we define **Accuracy** as: **(TP+TN) / (TP+TN+FP+FN)**



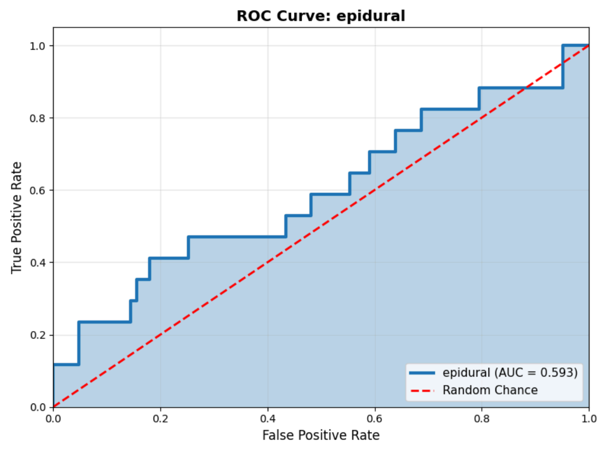


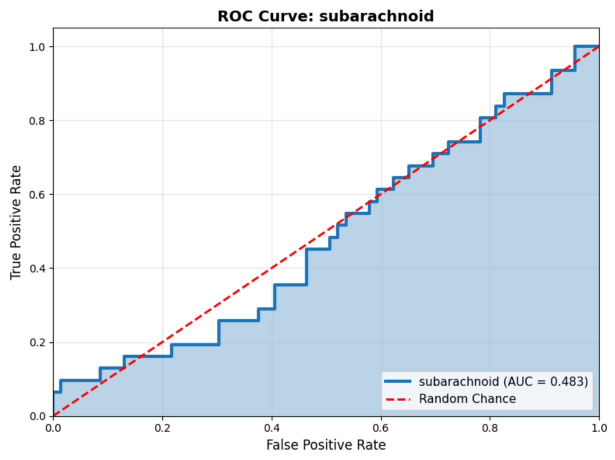


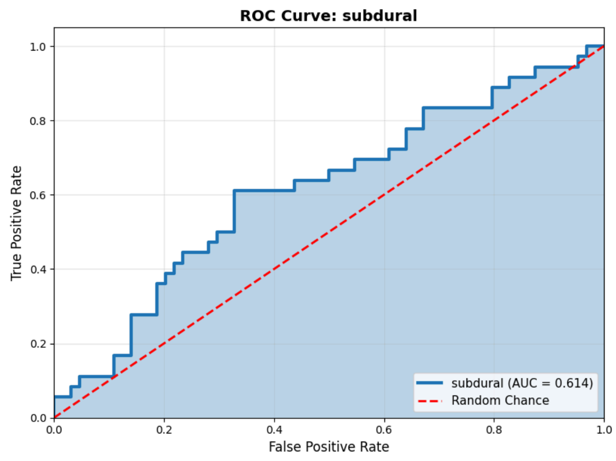


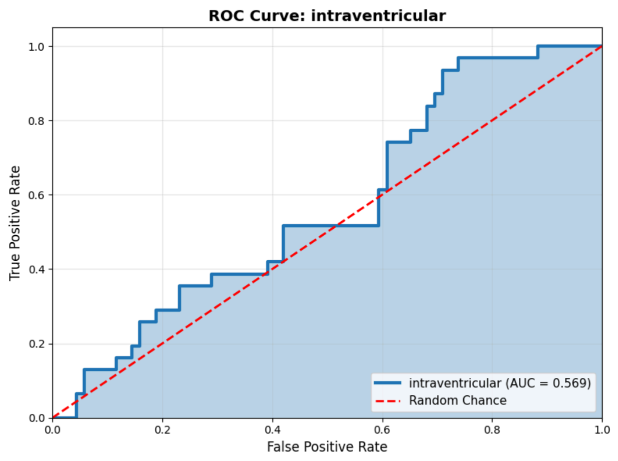


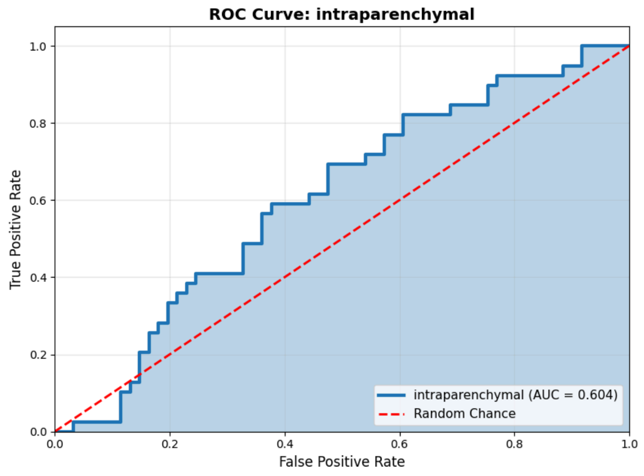
ROC-AUC or Receiver Operating Characteristic Area Under Curve, is another metric used to measure the model's ability to distinguish between positive and negative classes for binary classification by plotting values between **TPR (True Positive Rate)** and **FPR (False Positive Rate).** It ranges between 0 to 1, with 0.5 meaning it is as good as random guessing.











References:

1. Yeo, Melissa, Bahman Tahayori, Hong Kuan Kok, Julian Maingard, Numan Kutaiba, Jeremy Russell, Vincent Thijs et al. "Review of deep learning algorithms for the automatic detection of intracranial hemorrhages on computed tomography head imaging." Journal of neurointerventional surgery 13, no. 4: 369-378, 2021.

2. Flanders, Adam E., Luciano M. Prevedello, George Shih, Safwan S. Halabi, Jayashree Kalpathy-Cramer, Robyn Ball, John T. Mongan et al. "Construction of a machine learning dataset through collaboration: the RSNA 2019 brain CT hemorrhage challenge." Radiology: Artificial Intelligence 2, no. 3: e190211, 2020.
   Source: RSNA Brain Hemorrhage Classification Dataset

3. Yamashita, Rikiya, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. "Convolutional neural networks: an overview and application in radiology." Insights into imaging 9, no. 4: 611-629, 2018.

4. Sarvamangala, D. R., and Raghavendra V. Kulkarni. "Convolutional neural networks in medical image understanding: a survey." Evolutionary intelligence 15, no. 1: 1-22, 2022.

5. Yu, Hang, Laurence T. Yang, Qingchen Zhang, David Armstrong, and M. Jamal Deen. "Convolutional neural networks for medical image analysis: state-of-the-art, comparisons, improvement and perspectives." Neurocomputing 444: 92-110, 2021.

6. Rivas, Pablo. Deep Learning for Beginners: A beginner's guide to getting up and running with deep learning from scratch using Python. Packt Publishing Ltd, 2020.