21AIE312

Deep Learning for Signal & Image Processing

Project Report

Deep Learning for Health: Pneumonia Prediction

BATCH B GROUP 10

TEAM MEMBERS:

1. Seetharaman P – CB.EN.U4AIE21164
2. Maha Nivetha J K – CB.EN.U4AIE21132
3. Gayathri Devi R V – CB.EN.U4AIE21112
4. Devika K – CB.EN.U4AIE21108

Declaration

We declare that the report titled **“Deep Learning for Health: Pneumonia Prediction”** submitted by us is an original work done by Team-10 under the guidance of Dr. Mithun Kumar Kar, Associate Professor, School of Artificial Intelligence, Amrita Vishwa Vidyapeetham, Coimbatore during the sixth semester of the academic year 2023-24, in the School of Artificial Intelligence. The work is original and wherever we have used materials from other sources, we have given due credit and cited them in the text of the report. This report has not formed the basis for the award of any degree, diploma, associate-ship, fellowship or other similar title to any candidate of any University.

TEAM MEMBERS:

1. Seetharaman P – CB.EN.U4AIE21164
2. Maha Nivetha J K – CB.EN.U4AIE21132
3. Gayathri Devi R V – CB.EN.U4AIE21112
4. Devika K – CB.EN.U4AIE21108

Abstract

Pneumonia, a severe respiratory infection, poses a significant global health challenge, causing millions of deaths annually. Segmentation of pneumonia in medical imaging, particularly in chest X-rays and CT scans, is crucial for accurate diagnosis and effective treatment. Deep learning models have shown remarkable success in medical image analysis, offering promising tools for automated and precise segmentation. In this study, we employ three prominent deep learning architectures—UNet, VGG, and ResNet—to segment pneumonia-affected regions in medical images. Each model's performance is evaluated in terms of accuracy, computational efficiency, and robustness, providing insights into their suitability for clinical applications.

Pneumonia and Its Effects

Pneumonia is an inflammatory condition affecting the alveoli in the lungs, caused by infections or non-infectious factors. Symptoms include cough, fever, chest pain, and difficulty breathing. Severe cases can lead to complications like pleural effusion, lung abscess, and acute respiratory distress syndrome (ARDS), potentially resulting in respiratory failure and death. The burden is heavier in low-resource settings. Early and accurate diagnosis is crucial for effective management and treatment, reducing morbidity and mortality.

Segmentation of Pneumonia

Segmentation of pneumonia in medical images is critical for diagnosis. It involves identifying and delineating pneumonia-affected lung regions on chest X-rays or CT scans. Accurate segmentation helps assess the disease's extent and progression, aiding treatment planning and monitoring. Traditional image analysis relies on manual interpretation, which can be time-consuming and variable. Deep learning enables automated, consistent, and precise segmentation, enhancing diagnostic accuracy and efficiency.

Deep Learning Models for Segmentation

- In this study, we explore three well-established deep learning architectures—UNet, VGG, and ResNet—for the segmentation of pneumonia in medical images. Each model has distinct characteristics and strengths:
- UNet: Features a symmetric encoder-decoder structure, excelling in biomedical image segmentation.
- VGG: A deep convolutional network known for simplicity and strong feature extraction.
- ResNet: Utilizes residual learning for deep networks, addressing the vanishing gradient problem and learning complex patterns.

DATASET: -

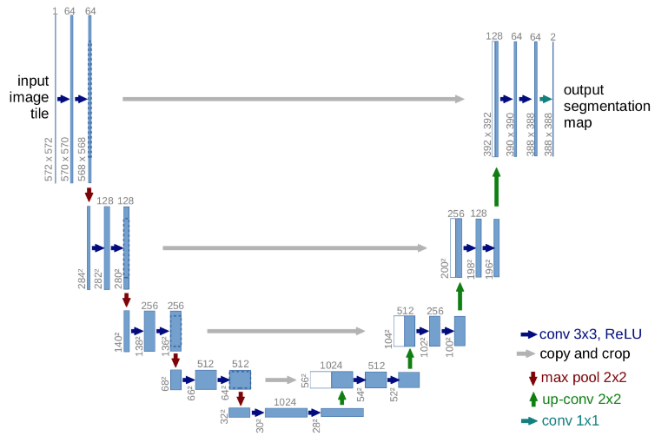
The RSNA dataset for pneumonia detection comprises approximately 30,000 images in .dcm format. The dataset includes separate train and validation sets, each containing around 27,000 and 4,000 images, respectively. The images are of 1024x1024 resolution. Additionally, a .csv file containing bounding box coordinates for pneumonia regions accompanies the dataset.

We have created mask images, representing pneumonia affected regions, are binary with a value of 1 inside the bounding box and 0 outside. They are converted to one-hot encodings

For the analysis of chest x-ray images, all chest radiographs were initially screened for quality control by removing all low quality or unreadable scans. The diagnoses for the images were then graded by two expert physicians before being cleared for training the AI system. In order to account for any grading errors, the evaluation set was also checked by a third expert.

MODELS:

UNET:



U-Net is a convolutional neural network (CNN) architecture designed primarily for biomedical image segmentation tasks. It was introduced by researchers at the Computer Science Department of the University of Freiburg. U-Net is particularly effective for tasks such as cell segmentation, organ localization, and medical image analysis. Its architecture is characterized by its symmetric encoder-decoder structure and the extensive use of skip connections.

U-Net Architecture Breakdown

Encoder:

The encoder part of U-Net captures features from the input image and extracts high-level representations. It typically consists of a series of convolutional and pooling layers arranged hierarchically.

- Convolutional Blocks: Each block usually comprises two or more convolutional layers followed by batch normalization and ReLU activation. The convolutional layers extract features, with the number of channels increasing with depth to capture increasingly abstract features.
- Pooling Layers: After each convolutional block, a max-pooling operation reduces the spatial dimensions of the feature maps, helping capture invariant features and reduce computational complexity.

Decoder:

The decoder part up-samples the feature maps from the encoder to generate the final segmentation mask, mirroring the encoder's structure but performing upsampling instead of downsampling.

1. Upsampling Blocks: Each block consists of an upsampling operation followed by one or more convolutional layers. The upsampling operation increases the spatial dimensions of the feature maps to match those from the encoder, while convolutional layers refine the feature representations.
2. Skip Connections:
    - Concatenation: Feature maps from the encoder are concatenated with corresponding feature maps from the decoder, allowing the decoder to access both low-level and high-level features, facilitating precise segmentation.
    - Preservation of Spatial Information: Skip connections help preserve spatial information during upsampling, enabling the network to generate accurate segmentation masks with well-defined boundaries.

Final Layer:

The final layer typically consists of a 1x1 convolutional layer followed by a softmax or sigmoid activation function, depending on the number of classes in the segmentation task. This layer produces the final segmentation mask with the same spatial dimensions as the input image.

Key Features:

- Symmetric Structure: The encoder and decoder have a symmetric structure, with each level of downsampling in the encoder corresponding to a level of upsampling in the decoder. This symmetry helps preserve spatial information and capture detailed features.
- Skip Connections: Extensive use of skip connections allows U-Net to effectively combine features from different levels of abstraction, leading to precise segmentation results with well-defined boundaries.

- Efficient Training: U-Net facilitates efficient training by leveraging pretrained weights from the encoder layers, reducing the need for large annotated datasets.

Overall, U-Net's architecture and design make it well-suited for a wide range of biomedical image segmentation tasks, where accurate delineation of structures and objects is crucial.

Data Preprocessing Steps:

- Installing and Importing Libraries: Essential libraries such as pydicom, opencv-python, and skimage are installed and imported. These libraries facilitate the handling and processing of medical images in DICOM format.

- Converting DICOM to JPG: The DICOM images are read and converted to JPG format, making them easier to handle and process. A directory is created for storing the JPG images, and each DICOM file is read, converted, and saved as a JPG image.

- Creating Mask Images: Masks are generated based on bounding box coordinates provided in a CSV file. The code reads the CSV file, checks for the presence of pneumonia, and draws bounding boxes on the masks accordingly. These masks are saved as JPG images for each corresponding input image.

- Loading Images and Masks into Arrays: The first 200 images are loaded, resized, and normalized. These preprocessed images are stored in an array, ready to be fed into the neural network for training.

U-Net Model Definition:

Custom Loss and Metrics: The Dice coefficient and Dice loss functions are defined to measure the segmentation performance. The Dice coefficient evaluates the overlap between the predicted and true masks, and the Dice loss is used as a loss function during training.

```
def dice_coef(y_true, y_pred):

  smooth=1.0e-6
  #y_true_f_1 = tf.keras.backend.flatten(y_true[:,:,:,0])
  #y_pred_f_1 = tf.keras.backend.flatten(y_pred[:,:,:,0])
  y_true_f_2 = tf.keras.backend.flatten(y_true[:,:,:,1])
  y_pred_f_2 = tf.keras.backend.flatten(y_pred[:,:,:,1])
  #intersection_1 = tf.keras.backend.sum(y_true_f_1 * y_pred_f_1)
  intersection_2 = tf.keras.backend.sum(y_true_f_2 * y_pred_f_2)
  #union_1 = tf.keras.backend.sum(y_true_f_1) + tf.keras.backend.sum(y_pred_f_1)
  union_2 = tf.keras.backend.sum(y_true_f_2) + tf.keras.backend.sum(y_pred_f_2)

  score = 2*(intersection_2+smooth)/(union_2+smooth)
  return score

def DiceLoss(y_true, y_pred):

  score = dice_coef(y_true, y_pred)

  return 1 - score

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (1,))

    ### [First half of the network: downsampling inputs] ###

    # Entry block
    x = layers.Conv2D(32, 3, strides=2, padding="same")(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    previous_block_activation = x  # Set aside residual

    # Blocks 1, 2, 3 are identical apart from the feature depth.
    for filters in [64, 128, 256]:
        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(filters, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    ### [Second half of the network: upsampling inputs] ###

    for filters in [256, 128, 64, 32]:
        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.UpSampling2D(2)(x)

        # Project residual
        residual = layers.UpSampling2D(2)(previous_block_activation)
```

```
        residual = layers.Conv2D(filters, 1, padding="same")(residual)
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    # Add a per-pixel classification layer
    outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

    # Define the model
    model = keras.Model(inputs, outputs)
    return model
cells were run multiple times
#keras.backend.clear_session()
```

Compiling the Model: The model is compiled with the Adam optimizer, Dice loss, Mean IoU, and Dice coefficient as metrics. These settings ensure that the model is trained effectively and its performance is evaluated accurately.

Training the Model:

Callbacks: Various callbacks are set up to monitor the training process. These include early stopping, model checkpointing, CSV logging, and TensorBoard logging. These callbacks help in saving the best model, stopping training early if necessary, and logging the training progress.

Training: The model is trained using data generators for training and validation. The training process is monitored using the specified callbacks to ensure that the model converges and performs well on the validation data.

Model Saving and Plotting Learning Curves

Saving the Model: The trained model weights are saved to a specified file for later use or further training.

Plotting Learning Curves: The learning curve showing the training and validation loss over epochs is plotted. This helps in understanding the model's performance during training and identifying any overfitting or underfitting issues.

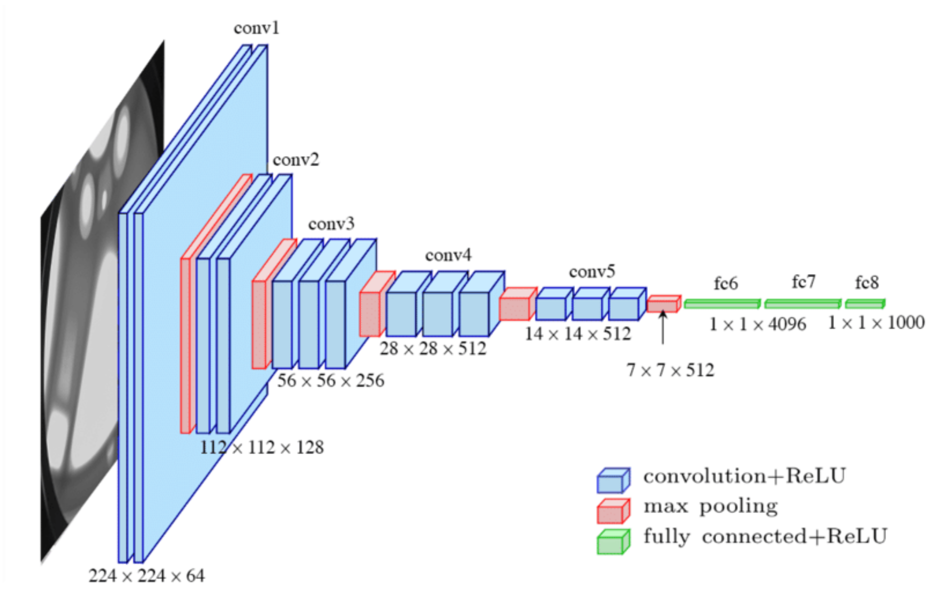
Plotting Dice Coefficient Curve: The Dice coefficient over epochs for both training and validation data is plotted. This metric indicates how well the model segments the images, providing insight into its segmentation performance.

Evaluating the Model

Model Evaluation: The model is evaluated on the validation data generator, and the performance metrics are printed. This step ensures that the model's performance on unseen data is acceptable and meets the desired criteria.

By following these steps and understanding the functions involved, you can preprocess medical images, define and train a U-Net model for segmentation tasks, and evaluate and visualize the model's performance. This comprehensive approach ensures that the model is trained effectively and its results are accurately assessed.

VGG16:



VGG16, developed by the Visual Geometry Group at the University of Oxford, is a convolutional neural network (CNN) architecture notable for its straightforward design and effectiveness in computer vision tasks. The architecture is characterized by its depth, consisting of 16 layers, including convolutional layers, pooling layers, and fully connected layers.

The network's architecture is built around repeated blocks of convolutional layers, each followed by a max-pooling layer. These convolutional layers use small 3x3 filters with a stride of 1 and a padding of 1, preserving spatial information while reducing the dimensionality of the input. The pooling layers, typically using 2x2 windows with a stride of 2, further reduce the spatial dimensions of the feature maps.

After several convolutional blocks, VGG16 includes three fully connected layers with ReLU activation. The first two fully connected layers have 4096 units each, followed by a third fully connected layer with 1000 units, corresponding to the number of classes in the ImageNet dataset, on which VGG16 was originally trained.

Throughout the network, ReLU activation functions are used to introduce non-linearity, helping the network learn complex patterns in the data. The use of small filters and deep architecture allows VGG16 to learn hierarchical features from images, leading to impressive performance in tasks such as image classification.

VGG16 is often used with pretrained weights, which are weights that have been trained on large-scale datasets like ImageNet. These pretrained weights can be fine-tuned on smaller, task-specific datasets, or used as feature extractors in transfer learning, where the learned representations are used to improve performance on new tasks.

Data Loading and Transformation:

Loading Data: The code uses PyTorch's ImageFolder dataset class to load images from directories (TRAIN, VAL, TEST) and applies transformations such as resizing, cropping, normalization, and converting to tensors using transforms.Compose.

Model Definition and Transfer Learning:

Loading Pretrained Model and Modifying Classifier: The code initializes a pretrained VGG16 model from PyTorch's models module and modifies the final fully connected layer to match the number of classes in the dataset. It also freezes the weights of the pretrained layers.

```
model_path = "/content/drive/MyDrive/vgg16-397923af.pth/vgg16-397923af.pth"
model_pre = models.vgg16()
model_pre.load_state_dict(torch.load(model_path))
num_features = model_pre.classifier[6].in_features
features = list(model_pre.classifier.children())[:-1]
features.extend([nn.Linear(num_features, len(class_names))])
model_pre.classifier = nn.Sequential(*features)

print(model_pre)

model_pre = model_pre.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model_pre.parameters(), lr=0.001, momentum=0.9,
weight_decay=0.01)
```

Training Loop:

Training Function: The code defines a training function (train_model) that iterates over the dataset for a specified number of epochs, updating the model weights based on backpropagation. It uses a specified loss function, optimizer, and learning rate scheduler.

Testing and Evaluation:

Testing Function: The code defines a testing function (test_model) that evaluates the trained model on a test dataset, calculating the accuracy of the model predictions compared to the ground truth labels.

Visualization:

Displaying Sample Images: The code includes a visualization function to display sample images from the dataset along with their predicted and actual labels, providing a visual assessment of the model's performance.

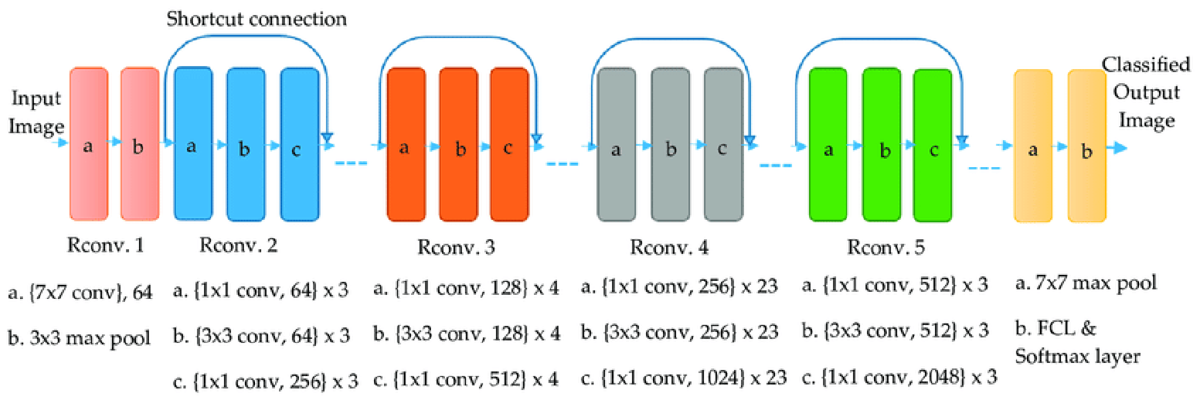
Optimizer, Learning Rate Scheduler, and Loss Function:

Optimizer: Stochastic Gradient Descent (SGD) optimizer is used with momentum and weight decay for better convergence and regularization.

Learning Rate Scheduler: The learning rate scheduler reduces the learning rate by a factor of 0.1 every 10 epochs to improve convergence.

Loss Function: Cross-entropy loss is used as the loss function for both training and testing.

RESNET:



ResNet-101:

ResNet-101 is a convolutional neural network architecture that builds upon the concept of residual learning to facilitate the training of very deep networks.

Architecture: -

Input Layer: The input to ResNet-101 is typically an RGB image of size 224x224 pixels.

Convolutional Layers: The network begins with a single convolutional layer followed by a max-pooling layer. This is followed by a series of residual blocks.

Residual Blocks: ResNet-101 comprises several residual blocks stacked on top of each other. Each residual block consists of multiple convolutional layers, usually with 3x3 kernels, and batch normalization followed by ReLU activation functions. The key innovation in ResNet is the introduction of shortcut connections. These connections skip one or more layers and directly feed the input to the output of the block. Mathematically, the output of a residual block is given by:

Output =ReLU(F(Input)+Input)

where,

F(Input) represents the output of the convolutional layers within the block.

Deep Architecture: ResNet-101 is exceptionally deep, consisting of 101 layers. The depth allows the network to learn intricate features from the input images, leading to superior performance in tasks such as image classification and object detection.

Global Average Pooling: After the stack of residual blocks, ResNet-101 typically uses global average pooling to reduce the spatial dimensions of the feature maps to a vector of size 1x1xN, where N is the number of channels (or features).

Fully Connected Layer: Finally, a fully connected layer with softmax activation is used to produce the final output probabilities for classification tasks.

Data Loading and Preprocessing:

The dataset is loaded using datasets.ImageFolder and transformed using transforms.Compose for both training and testing datasets. This ensures that the images are resized, cropped, converted to tensors, and normalized before being fed into the model.

Model Initialization:

The ResNet-101 model is loaded from torchvision.models and the final fully connected layer (fc) is modified for binary classification. The rest of the model weights are frozen to retain the pre-trained knowledge.

model_path = "/content/drive/MyDrive/resnet101-5d3b4d8f.pth/resnet101-5d3b4d8f.pth"

model_pre = models.resnet101()

model_pre.load_state_dict(torch.load(model_path))

```
for param in model_pre.parameters():
    param.requires_grad = False


model_pre.fc = nn.Sequential(
            nn.Linear(2048, 128),
            nn.ReLU(inplace=True),
            nn.Linear(128, 2)).to(device)
print(model_pre)
```

Training: The train_model function is defined to train the model. It iterates over the specified number of epochs, performing forward and backward passes, updating weights, and evaluating the model's performance on the validation set. The learning rate is adjusted using a scheduler.

Testing: The test_model function is defined to evaluate the trained model on the test set. It calculates the accuracy of the model on the test set and returns the true labels, predicted labels, input images, and accuracy.

Model Evaluation: The trained model is evaluated on the test set, and the accuracy and some sample predictions are displayed using matplotlib.

Parameters:

Loss Function: CrossEntropyLoss

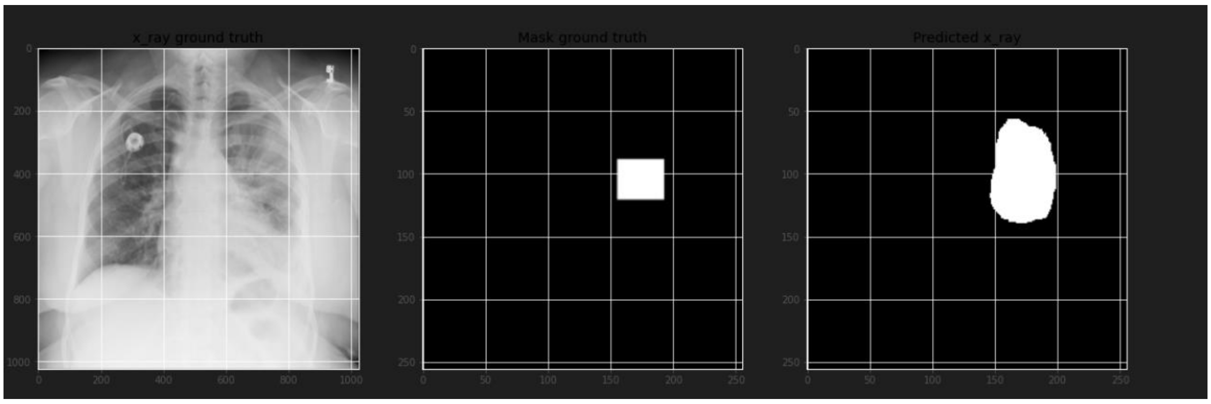
Optimizer: Stochastic Gradient Descent (SGD) with momentum and weight decay

Learning Rate: 0.001

Scheduler: StepLR with step size 10 and gamma 0.1

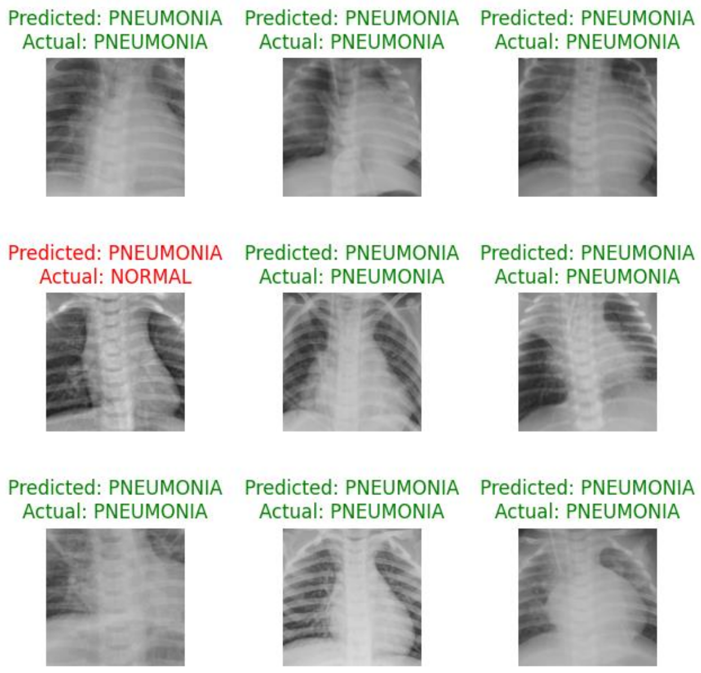
RESULTS:

UNET:



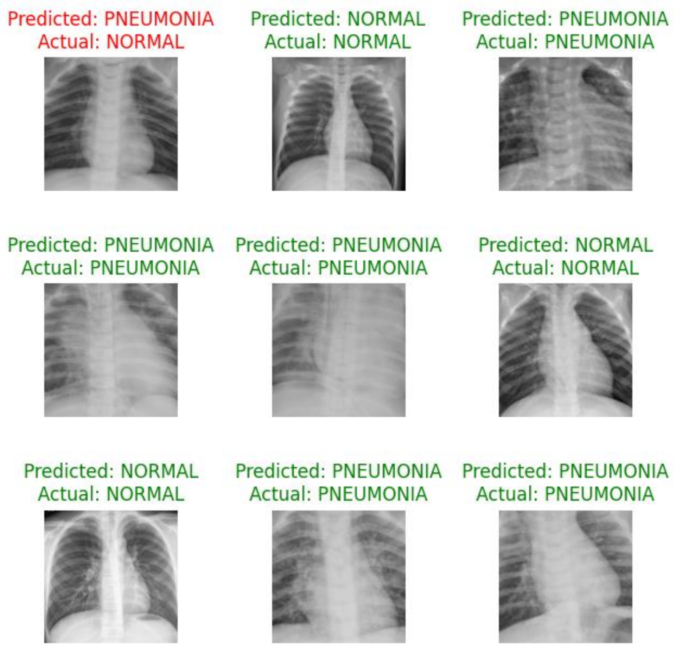
The model achieved a dice coefficient of 0.95 and mean iou of 0.9528 and the results were correctly predicted.

VGG:



The VGG16 model achieved an accuracy of 83.9% with most of the labels being correctly predicted

RESNET:



The model achieved an accuracy of 84.6 being the highest among all and labels being correctly predicted

REFERENCES: -

1. https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47
2. U-Net: Convolutional Networks for Biomedical Image Segmentation
3. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9759647/
4. VGG16 Model for Pneumonia Image Classification
5. https://www.researchgate.net/publication/340096057_Effective_Pneumonia_Detection_using_ResNet_based_Transfer_Learning