Image Classification of the Kvasir Multi-Class Image Datasets Using Deep Learning and Machine Learning Techniques

Babirye Daphne

Data Analytics

Table of Contents

Table of Figures

Table of Tables

1.0.Introduction

The report analyses two versions of a Multi-Class Image-Dataset for Computer Aided Gastrointestinal Disease Detection; the Kvasir Dataset and applies two different machine learning techniques. The details of the results are presented in the subsequent sections of the report

1.1...Objective of report

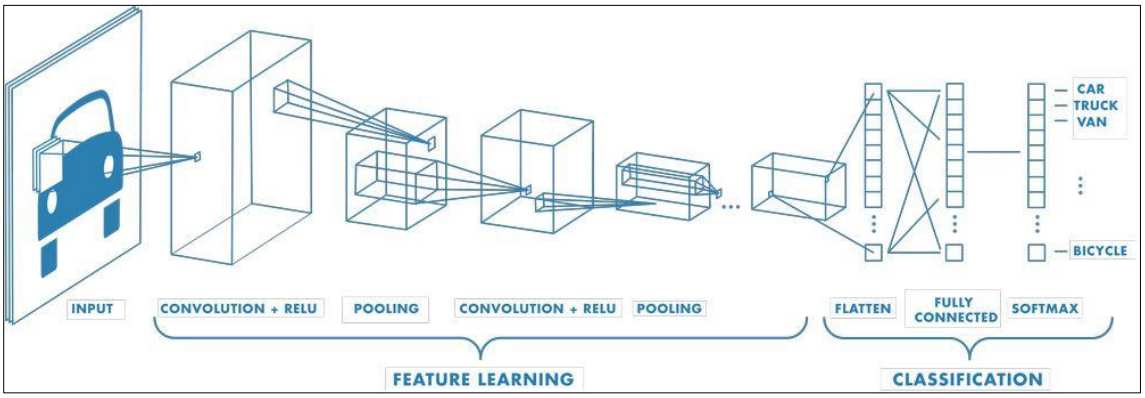
The objective of the Assignment is to undertake Analysis on 2 datasets using Convolution Neural Networks and Statistical Learning techniques

2.0.Convolutions

A Convolutional neural network (CNN) is one of the techniques under deep learning that is useful in a number of tasks including classification. A CNN is composed of a number of layers, such as convolution layers, pooling layers, and fully connected layers, that are used to learn and detect the different features of an image in the case of image classification. However, they are also effective for classifying audio, time-series, and signal data (Yamashita et al., 2018).

Filters are applied to each image at different resolutions during training, and the output of each convolved image is an input in the next layer. The initial filters identify simple features, such as brightness and edges, and the later filters detect features of higher complexity that uniquely define the object such as shapes. **Figure 3** displays an example of a CNN architecture(The MathWorks Inc, 2023).

Figure 1: Diagram of a convolution and the Architecture of a Convolution Neural Network



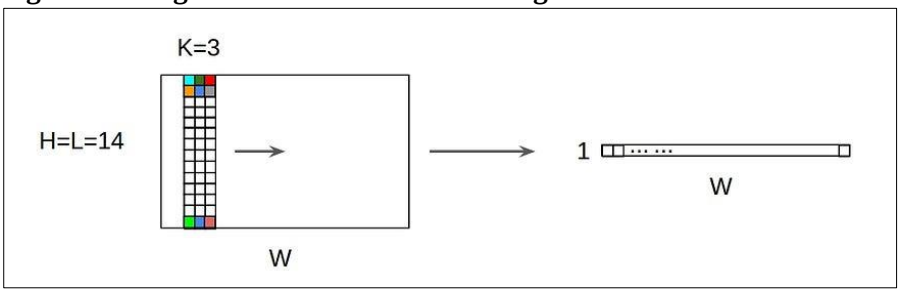
Source: https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html

2.1.Explanations of the Workings of Convolutional Neural Networks

2.2.2D input image with a 1D conv operation

For a 2D input image we assume a grayscale images and we are interested in patterns that occur along the horizontal axis of the image such as an edge or a pattern. A 1D convolution can be used along the horizontal axis. The process would entail defining a 1D filter or kernel which is a 1D array of weights. The kernel is moved along the horizontal axis of the image, and at each step, it performs an element-wise multiplication with the part of the image it's currently on, and then sums up the results to give a single output value (runhani, 2017). **Figure *1*** presents an image of the operation.

Figure 1: Image of a 1D Conv on a 2D Image



Source: https://stackoverflow.com/questions/42883547/intuitive-understanding-of-1d-2d-and-3d-convolutions-in-convolutional-neural-n/44628011#44628011

In terms of a full 1D conv on a 2D image model, suppose we have 32 filters, a kernel size of 3, and the input shape is a 2D array with size (100, 10), where 100 is the length of the time dimension and 10 is the number of features. Table 1 presents the full model.

Conv1D layer: This layer performs a 1D convolution operation. It applies 32 filters of size 3 to the input. The input to this layer is a 2D array with shape (100, 10). The layer slides each filter across the time dimension of size 100 of the input, performing an element-wise multiplication and sum at each step. This results in a single value for each position of the filter. The process is repeated for each of the 32 filters. The output of this layer will have shape (98, 32), because each end of the time dimension loses one position due to the size of the filters. The activation function 'relu' is applied to the output, which introduces non-linearity and helps the network learn more complex patterns.

Flatten layer: This layer reshapes the input into a 1D array. It doesn't perform any computations and doesn't have any parameters.

Dense layer: This is a fully connected layer, which means each input is connected to each output by a weight parameter. In this case, we have one output unit, so this is a binary

classification task. The 'sigmoid' activation function is applied, which squashes the output to a range between 0 and 1.

Finally, the model is compiled with the binary cross entropy loss function, which is suitable for binary classification, and the Adam optimizer, which is a commonly used optimization algorithm.

In terms of parameters, the Conv1D layer a 1D convolutional layer with 32 filters and a kernel size of 3. The input has 10 channels, so each filter must have a set of weights for each channel. This gives us 3 (weights per filter per channel) * 10 (channels) * 32 (filters) = 960 weights. Additionally, each of the 32 filters has a bias term, giving us 32 biases. So in total, this layer has 960 weights + 32 biases = 992 parameters.

The flatten layer does not have any weights or biases. It simply changes the shape of the output from the previous layer. It therefore has 0 parameters.

The Dense layer is a fully connected layer, meaning each input is connected to an output by a weight. The input to this layer has 98 (the output size of the Conv1D layer after sliding the 3-sized kernel over the 100-sized input) * 32 (number of filters in the Conv1D layer) = 3136 units. This layer has 3136 (inputs) * 1 (outputs) = 3136 weights. Additionally, there is 1 bias term for the output node, giving 1 bias. In total, this layer has 3136 weights + 1 bias = 3137 parameters.

All the parameters from all layers, give a total of 992 (Conv1D) + 0 (Flatten) + 3137 (Dense) = 4129 parameters as seen in *Table 1*.

Table 1: Model Parameters for a 1D Conv on a 2D Image CNN

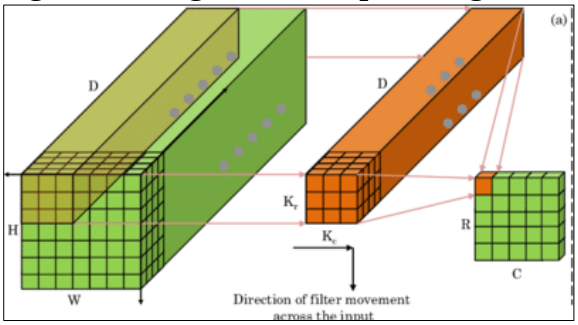
```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_2 (Conv1D)           (None, 98, 32)            992

 flatten_10 (Flatten)        (None, 3136)              0

 dense_26 (Dense)            (None, 1)                 3137

=================================================================
Total params: 4,129
Trainable params: 4,129
Non-trainable params: 0
```

2.3.3D input with a 2D conv

RGB images consist of three channels, namely; Red, Green, and Blue, making them a 3D input, as it has a height, width, and depth. A 2D convolution operation implies that the convolutional filter/kernel moves in two dimensions which are height and width across an image, as it moves across the entire image at each step as seen in **Figure *2*** (Mittal & ., 2021).

The kernel is a small matrix of weights that slide over the input data, performing an elementwise dot product with the part of the input image it is currently on, and thereafter sums up the results into one output pixel. This operation is performed in every location the kernel can fit on the input data, resulting in a new output feature map(Brownlee, 2019).

Figure 2: Image of a 3D input Image and CONV 2D operation



Source: https://www.researchgate.net/figure/a-2D-CONV-on-3D-input-The-filter-moves-only-in-two-directions-height-and-width-of-the_fig1_348805304

An example of 3D input and 2Dconv model consists of a single 2D convolutional layer that takes as input a 64x64 pixel RGB image (hence the input_shape=(64,64,3)). The model uses 32 filters, meaning it will output 32 feature maps, each of which is a 3x3 kernel. The stride of 1 means the filter will move 1 pixel at a time as it scans the image. The 'valid' padding means there is no padding added, and the filter won't go outside the image boundaries.

For a full model one would include a pooling layer (like MaxPooling) after the Conv2D layer, a flattening layer before a Dense layer, and output layer that matches the task at hand (for instance, for a 10-class classification, a Dense layer with 10 units and a 'softmax' activation function).

Calculation of parameters

In the Conv2D layer, the number of parameters is determined by the number of filters and the size of each filter, along with the number of input channels to the layer as seen in ***Table 2***.

Each filter has a size of 3x3 as specified by kernel_size=(3,3), and the input has 3 channels as we're using RGB images. Each filter therefore has 3x3x3 = 27 weights. Each filter also has a bias term. So, the total number of parameters per filter is 27 weights plus 1 bias term = 28 parameters. Since we have 32 filters as specified by filters=32, the total number of parameters in the Conv2D layer is 32 filters x 28 parameters/filter = 896 parameters.

Table 2: Parameters of a 3D input Image and CONV2D Operation

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_8 (Conv2D)           (None, 62, 62, 32)        896

=================================================================
Total params: 896
Trainable params: 896
Non-trainable params: 0
```

2.4.2D input with a 3D conv

A 3D convolution is an extension of a 2D convolution which adds an additional depth dimension in the input data and the convolution kernel. This extra dimension represents different things depending on the context of the data. For example, in medical imaging, the depth could correspond to the z-axis of a 3D scan. In a 3D convolution, you also slide the kernel across the depth of the input, implying that the kernel is not only looking for patterns in a 2D space, but also across different depths(CS231n, 2023).

A 3D convolution operation would need three spatial dimensions, but if we have a 2D input, the channels dimension can be treated as a depth, making the input seem like 3D. For example, an RGB image of size 64x64, could be treated as a 3D input of size 64x64x3.

Table 3 displays a model with Conv 3D and 2D input. The Conv3D layer is a 3D convolution layer with 32 filters and kernel size of 3x3x3, with a 64x64 RGB image as a 64x64x3 input, where each color channel is treated as a separate depth portion. The kernel will convolve across height, width and also across the channels. The Flatten layer reshapes the 3D output from the previous layer into a 1D array so that it can connect to the Dense layer. The dense layer is a fully connected layer with 10 nodes, one for each class we want to predict. The softmax activation function is used to ensure the output is a probability distribution over the 10 classes.

Parameters

Conv3D Layer: The number of parameters for a Conv3D layer can be calculated as (number of filters x volume of each filter x number of input channels) + number of filters for the bias terms. In this case, we have 32 filters each of size 3x3x3 and the input has 1 channel. So the total parameters for the Conv3D layer would be (32 x 3 x3 x 3 x 1) + 32 = 896.

Flatten Layer: The Flatten layer doesn't have any weights or biases. It doesn't contribute any parameters.

Dense Layer: The number of parameters for a Dense layer can be calculated as (number of inputs x number of outputs) + number of outputs for the bias terms. Assuming the padding in the Conv3D layer is valid , the output size from the Conv3D layer would be (62, 62, 1, 32), resulting in 62x62x1x32 = 123008 inputs. Since we have 10 output nodes, the total parameters for the Dense layer would be (123008 x 10) + 10 = 1230090.

Adding these together, the total number of parameters in the model is 896 (Conv3D) + 0 (Flatten) + 1230090 (Dense) = 1230986 trainable parameters.

Table 3: A Keras Model Summary of a 3D input and 2D Convolution

```
Model: "sequential_15"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv3d_2 (Conv3D)           (None, 62, 62, 1, 32)     896

 flatten_18 (Flatten)        (None, 123008)            0

 dense_34 (Dense)            (None, 10)                1230090

=================================================================
Total params: 1,230,986
Trainable params: 1,230,986
Non-trainable params: 0
```

3.0.Deep Learning Using the Kvasir-V2 dataset

3.1.Split the dataset random into train/validation/test 70:15:15

To split the data into train, validation and tests in the ratio 70:15:15, the code on github was used as reference with soe modification. I imported the necessary libraries including random, numpy, glob, shutil, TensorFlow and matplotlib. I then set seed to 283 for reproducibility of the data.

The directory that provides the path to the directory where the Kvasir-V2 dataset is stored was defined . A list of classes that represent the different types of images in the dataset was also defined. The dataset is split into a training set validation set and a test set in a ratio of 70:15:15.

For each class in the list of classes, the code creates a path to the directory containing the images for that class and uses the glob library to get a list of all the images in that directory. The number of images in the class is printed, and then the images are split into the specified lists. The images in the training list are then moved to a new directory called 'train' within the base directory. If the directory for that class does not exist in the 'train' directory, it is created using the os.makedirs() function. The same process is applied to the test list and valid list, therefore creating a valid and test directory/folder.

3.2.Challenges in configuration and environment setup and how they were addressed

Downloading the large dataset: The Kvasir-V2 dataset is a large dataset, and downloading took a significant amount of time, due to a slow internet connection. This was resoled by downloading the dataset using the university wifi which was more efficient.

Low storage space: The Kvasir dataset is large (2.3 GB), which took up a lot of storage space on my local machine to store the entire dataset. I stored it on the machine regardless of the size since I needed to perform some of the analysis using the local machine. However, I couldn't run the model on the machine as a GPU was required which the local machine did not have.

Setting up a virtual environment to run the model: Setting up the environment to work with the Kvasir-V2 dataset was challenging and time consuming as the computers in the lab had to be set up several times before they could actually work. I sought help from the IT personnel who were very helpful and always ready to provide assistance. However, this interrupted the time for working on the assignment.

3.3.Baseline model

3.3.1. Model Building

To create the model, key libraries were imported into the jupyter notebook including; keras from tensorflow, ImageDataGenerator, Sequential from models, Conv2D, MaxPooling2D, Dropout,Flatten,Dense, BatchNormalization from keras.layers, ModelCheckpoint and EarlyStopping keras callbacks, load_model from keras.models and matplotlib.pyplot as plt.

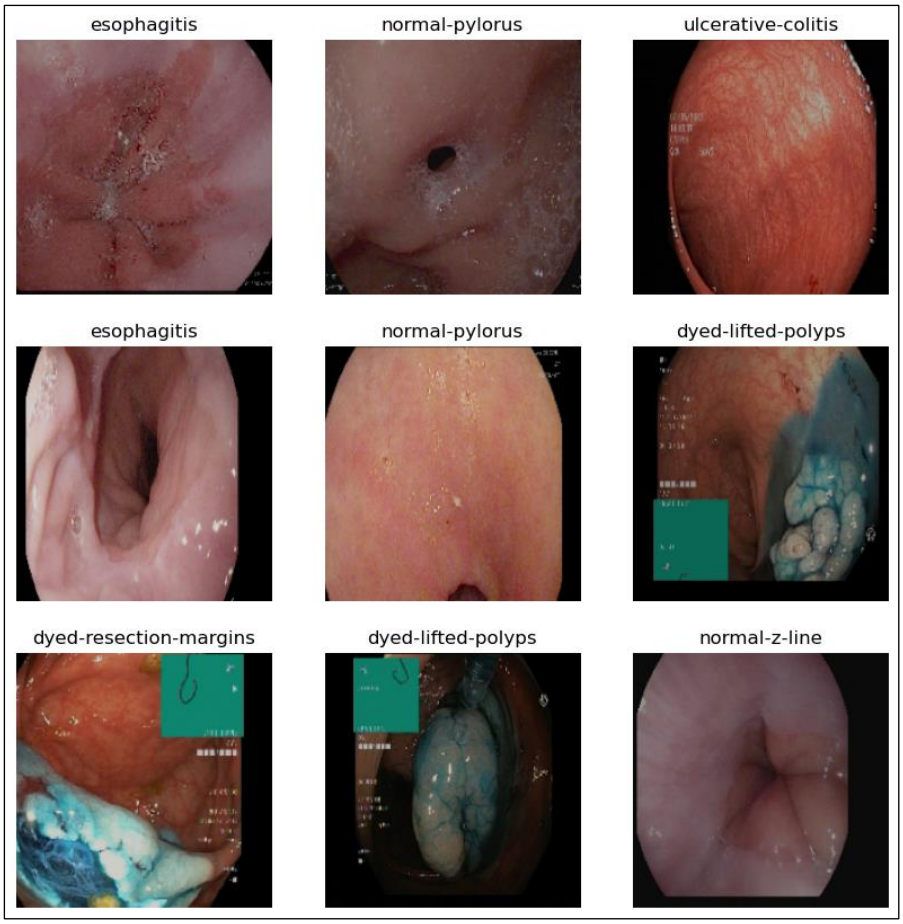
Data Augmentation; Data Augmentation involves techniques used to enhance the size and quality of training datasets to improve performance of Deep Learning models. They are also a solution to the limited data(Shorten & Khoshgoftaar, 2019). For the ksavir data set the following data augmentation techniques were applied to the training data; It specifies various augmentation techniques like shifting, zooming, flipping, brightness changes, and rescaling the pixel values between 0 and 1. No augmentation is applied to the validation and testing data.

Data generators for training, testing, and validation data were created. The data generators help to efficiently load and pre-process the data in batches, which allows the model to process the data in smaller parts during training, testing, and validation process.

The train data generator generated batches of augmented training data and read the images from the kvasir dataset directory, resized them to (200, 200) pixels, applied augmentation techniques, and provided the images and their corresponding labels as batches during training. Similarly, the test and valid data generators generated batches of test data. It read the images from the kvasir dataset directory, resized them to (200, 200) pixels, and provide the images and their corresponding labels as batches during validation and testing.

The train dataset had 5600 images belonging to the 8 classes while the validation and test data each had 1200 images belonging to the 8 classes. The train data class indices were as follows; 'dyed-lifted-polyps': 0, 'dyed-resection-margins': 1, 'esophagitis': 2, 'normal-cecum': 3, 'normal-pylorus': 4, 'normal-z-line': 5, 'polyps': 6, 'ulcerative-colitis': 7. **Figure *3*** below shows the images of the respective 8 classes of images.

Figure 3: Visualisation off the 8 Classes of Images



3.3.2. Comparison of chosen Baseline Model and Git hub Baseline Model

Model Architecture

Depth: The model from GitHub is deeper, with 6 convolutional layers, followed by 3 dense layers. The baseline model (model1) is less deep, with only 2 convolutional layers, followed by 2 dense layers.

Filter numbers: The github model uses more filters in the convolutional layers, starting from 32 and going up to 256, which allows it to extract a wider variety of features, while the baseline model starts with 16 filters and goes up to 32.

Dropout: Both models use dropout for regularization, but the locations are different. The github model applies dropout after flattening, while the baseline model applies dropout after MaxPooling and also after the first dense layer.

Dense Layers: The git hub model uses more nodes in dense layers (1024 and 512), which can capture more complex patterns but also increases the risk of overfitting whereas the baseline model uses fewer neurons (256 and 128).

MaxPooling: Both models use MaxPooling2D after each convolutional layer to reduce dimensionality, but the first model has more MaxPooling layers due to having more convolutional layers.

Overall, the git hub model is more complex and has a larger capacity to learn from data due to more filters and nodes. However, this also implies that it is more prone to overfitting and requires more computational resources to train. To the contrary, the baseline model is simpler and might be faster and less prone to overfitting. Nonetheless, it might perform worse for complex tasks. However, since it is baseline model, it is better for it to be simple as it will be improved to improve performance.

Explanation of each layer of the model network

Conv2D (16, (3, 3), activation='relu', padding='same', input_shape=(200, 200, 3)): This is the first layer of the model that interacts directly with the images. It's like a set of 16 different customizable filters that slide over the image and detect different features like edges, corners, and textures. The 'relu' activation function makes sure that if the filter doesn't detect a feature, it will output zero.

Batch Normalization: This layer helps the model learn by making sure the outputs from the previous layer have a standard distribution (like an average of 0 and standard deviation of 1). This makes the learning process more stable and efficient.

MaxPooling2D(pool_size=(2,2)): This layer reduces the image size by half in both dimensions by taking the maximum value in each 2x2 area. It helps to retain the important features while reducing the computational load for the next layers.

Conv2D (32, (3, 3), padding='same', activation='relu'): This layer is like a more complex set of 32 filters that can detect higher-level features in the images. These could be things like parts of objects or specific shapes.

BatchNormalization(): This helps to standardize the outputs for more stable and efficient learning as indicated earlier.

MaxPooling2D(pool_size=(2, 2)): This is another pooling layer that further reduces the size of the feature maps while retaining the most important features.

Flatten(): This layer transforms the 2D feature maps into a 1D array, so that the data can be fed into the fully connected layers that follow.

Dropout(0.2): This layer randomly sets 20% of the input units to 0 at each update during training time, which helps prevent overfitting.

Dense(256, activation='relu'): This is a fully-connected layer where each node is connected to every node in the previous layer. It learns to interpret the features extracted by the convolutional layers and makes classification decisions based on them.

Dropout(0.2): This layer helps prevent overfitting by randomly setting 20% of the input units to 0 at each update during training time as mentioned earlier.

Dense(128, activation='relu'): This is another fully-connected layer that continues the interpretation of the features and decision-making process to classify the images.

Dense(8, activation='softmax'): This is the final layer of the model, which makes the final decision about what class the input image belongs to. The 'softmax' activation function ensures that the outputs sum to 1 and can be interpreted as probabilities for each class.

This model is trained using 'adam' optimizer which is a method that changes the weights and biases in the model to minimize the error. The model's performance is evaluated using accuracy (the proportion of images correctly classified). During training, the model saves the best performing version of the model and stops training if it doesn't see improvement in validation loss for 5 epochs (a complete pass through the entire training dataset).

3.3.3. Performance of Base Model

Due to the limited time and capacity of the computers, 10 epochs were run to get an initial insight into the performance of the model.

With regard to training loss; the training loss started from a high value of 3.6282 and significantly dropped to 1.3048 in the second epoch. After the initial drop, the decrease in loss is gradual and reaches 0.9420 by the 10th epoch. This trend indicates that the model is learning from the training data.

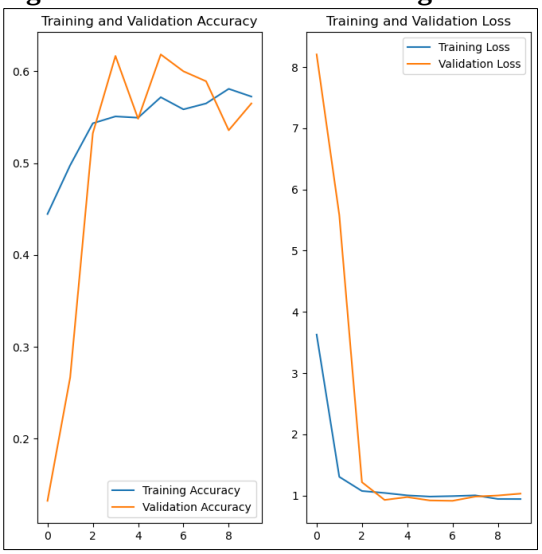
The validation loss started very high at 8.2048, dropped significantly in the second epoch, and then reduced to 0.9132 by the 7th epoch. However, it started increasing after the 7th epoch, reaching 1.0304 at the 10th epoch. This increase in validation loss towards the later epochs might suggest overfitting, as the model might be learning the training data too well and performing less optimally on new, unseen data.

The model's accuracy on the training data started at 44.46percent and increased slightly over time, reaching 57.25percent by the 10th epoch. The gradual increase in accuracy indicates that the model is improving its predictions on the training data over time.

The validation accuracy began low at 13.25percent but jumped to 53.25percent by the 3rd epoch. It reached its highest at 61.83percent in the 6th epoch, but then it fluctuated and dropped to 56.50 percent by the end of the 10th epoch.

Overall, the model is learning and improving its performance on the training set over time, but it's not performing as well on the validation data set, especially in the later epochs (**Figure *4***). The validation loss increases, and the accuracy decreases towards the end of training, which may indicate overfitting. This might suggest that the model is too complex or the training data may not be representative enough of the data in the validation set. To improve this performance, other models will be implemented with some adjustments in parameters.

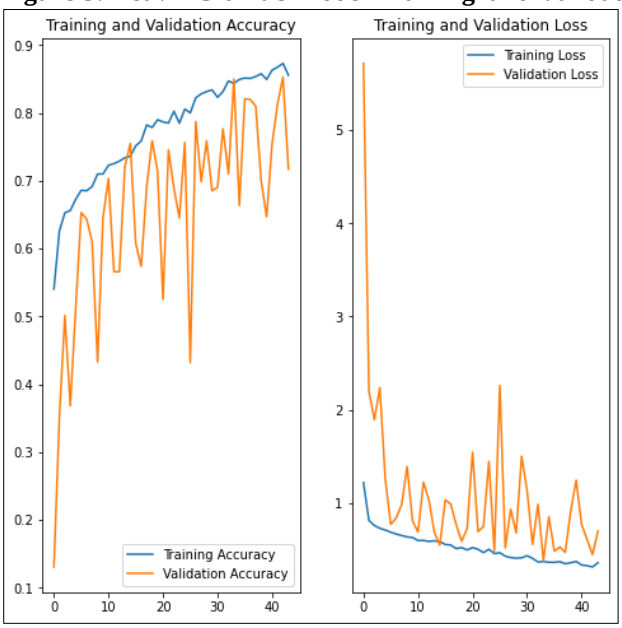
Figure 4: Baseline Model Training and Validation Curves



In comparison to the git hub model on the Ksavir dataset, the training and validation loss both decline and stabilise at a lower level as compared to the baseline model used in this report

(**Figure 5**). However, the training and loss steadily decline in the baseline model as compared to the git hub model where the validation loss keeps rising and declining, which are possible signs of over fitting. None the less both models seem to indicate that the models are learning well. With regard to accuracy and validation accuracy for the git hub model rise higher to approximately 85 percent while the baseline model training and validation accuracy rise to approximately 57 percent. This is likely due to the fact that the baseline model used here is a simpler model with less layers and nodes, and trained on 10 epochs compared to 100 epochs in the git hub model.

Figure 5: Ksavir Git Hub Model Training and Validation Curves



Source: https://github.com/AfraHussaindeen/Kvasir-Dataset/blob/master/Novel_CNN.ipynb

3.4.Performance of Tuned Models

3.4.1. Modified Model2

A number of changes were made to model 2 in order to improve model performance. Firstly, the modified model (model2) has two additional convolutional layers compared to the baseline model (model1); Baseline Model (model1) has 2 convolutional layers while modified model (model2) has 4 convolutional layers.

Secondly, the modified model (model2) has an increased number of filters in the additional convolutional layers, allowing it to capture more complex patterns and features in the images; the baseline Model (model1) has 16 filters in the first convolutional layer and 32 filters in the second convolutional layer while the modified Model (model2) has 16 filters in the first

convolutional layer, 32 filters in the second convolutional layer, 64 filters in the third convolutional layer and 128 filters in the fourth convolutional layer.

Lastly, the modified model (model2) has an increased number of nodes in the fully connected layers, which provides the model with more capacity to learn and represent more complex features in the data; the baseline Model (model1)has 256 nodes in the first fully connected layer and 128 nodes in the second fully connected layer while the modified Model (model2) has 512 nodes in the first fully connected layer and 256 nodes in the second fully connected layer.

Model Performance

Overall, the modifications made in the model architecture, such as increasing the number of convolutional layers, filters, and nodes in the fully connected layers, improved the model's performance. The model was able to capture more complex features, resulting in improved accuracy and lower loss values. The training curves show that the modified model achieves better performance compared to the baseline model (**Figure 6**).

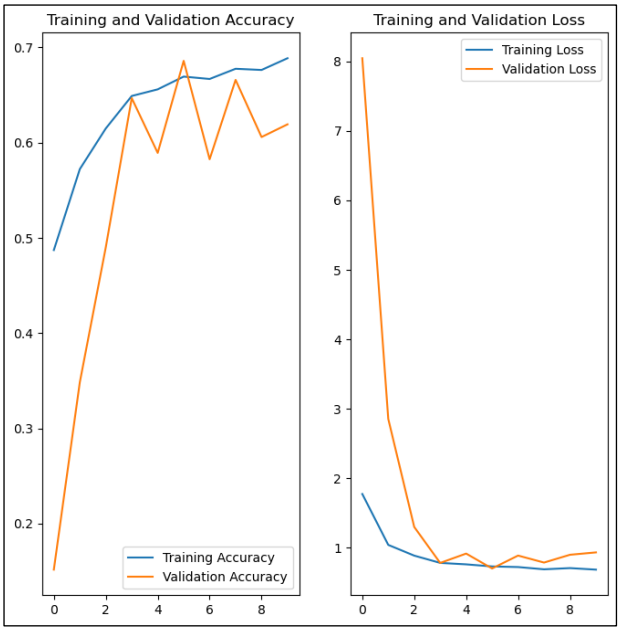
In the modified model (model2), the training loss decreases more gradually and reaches a lower value of 0.68 compared to the baseline model (model1) (0.94). This indicates that the modified model is better at fitting the training data and capturing its underlying patterns.

In addition, the modified model achieves a lower validation loss throughout the training process compared to the baseline model This suggests that the modified model generalizes better to unseen data and is less prone to overfitting.

The modified model also achieves a higher training accuracy of 0.69 compared to the baseline model 0.57. This indicates that the modified model is better at correctly classifying the training examples.

Lastly, the modified model also achieves a higher validation accuracy of 0.61 compared to the baseline model with 0.56 . This suggests that the modified model performs better on unseen data.

Figure 6: Model 2 Curves for Training and Validation Accuracy and Loss



3.4.2. Modified Model 3

Since there was a possibility of overfitting, the baseline model was adjusted by tuning the following parameters without increasing the layers of the model which could increase overfitting.

Dropout: Increased the dropout rate from 0.2 to 0.3. Dropout randomly sets a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting.

Early Stopping: Increased the 'patience' parameter from 5 to 10. This gives the model more room to improve before stopping training.

Epochs: Increased the number of epochs from 10 to 20. With the adjustments to early stopping, the model may benefit from more training rounds.

Model Performance

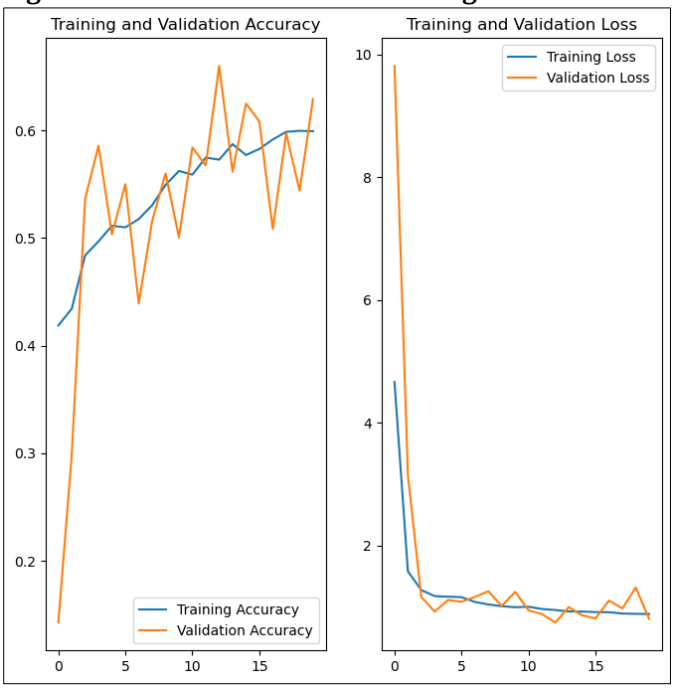
Overall, the changes made in the modified model, such as increasing the dropout rate, adjusting the early stopping parameters, and increasing the number of epochs, contributed to its improved performance. These changes helped the model reduce chances of overfitting, allowed for more training, and improved accuracy on unseen data. The training curves demonstrate that the modified model achieves lower validation loss and higher validation accuracy, indicating its enhanced capability of the model to recognize patterns and classify new images accurately (***Figure 7***).

In particular, the modified model (model3) shows a gradual decrease in training loss over the epochs. This indicates that the model is effectively learning from the training data and reducing its training error. The model also achieves a lower validation loss of 0.80 compared to the baseline model (model1) with 1.03. This improvement suggests that the model is better at generalizing to unseen data and has reduced overfitting.

In addition, the modified model (model3) achieves a similar training accuracy (0.59) compared to the baseline model (model1) (0.57.7). This suggests that both models can classify the training examples with a similar level of accuracy.

Lastly, the modified model (model3) achieves a higher validation accuracy (0.63) compared to the baseline model (model1) with 0.57. This improvement indicates that the modified model performs better on unseen data and has enhanced generalization capabilities.

Figure 7:Model 3 Curves for Training and Validation Accuracy and Loss



Final Model Evaluation

Based on the performance of the three models model 2 has the best performance with a training and validation accuracy of 0.69 and 0.62 respectively. Evaluation was therefore conducted on model, achieving a model Loss of 0.72 and model accuracy of 0.66.

However, with training over more epochs, more layers and adjusting additional parameters, higher improvement could be achieved.

3.5.Transfer Learning – VGG-16 model

3.5.1. Transfer Learning

Transfer learning is a machine learning method that uses knowledge gained from one task to improve performance on another related task. (Wikipedia, 2023). Similarly, in machine learning models, transfer learning involves the use of a pre-trained model as the source of knowledge to implement a modelling task on another dataset. The pre-trained model is usually trained on a large dataset for a related task, such as image classification on a large collection of images where it learns to recognize general features and patterns that are useful for the task. When applying transfer learning, the pre-trained model's knowledge, which is represented by its learned weights, is utilized as a starting point for training a new model on a related task. The first layers of the pre-trained model, which capture low-level and general features, are usually kept fixed, while the later layers are fine-tuned or replaced for adaptation to the new task(Jessica Powers, 2022).

3.5.2. Transfer learning Model

The VGG16 model is one of the best pre-trained models on the ImageNet dataset, and it's good for a wide range of image recognition tasks. However, the VGG16 is deeper and has more parameters than other models such as the ResNet50, which might make it slower to train and more prone to overfitting.

The new model (model4) is based on the pre-trained VGG16's model architecture which is a deep convolutional neural network (CNN) known for its effectiveness in image classification tasks. The base layers of the VGG16 model are frozen to prevent their weights from being updated during training, and new layers are added on top to adapt the model to the current classification task(www.keras.io, 2023).

Model performance

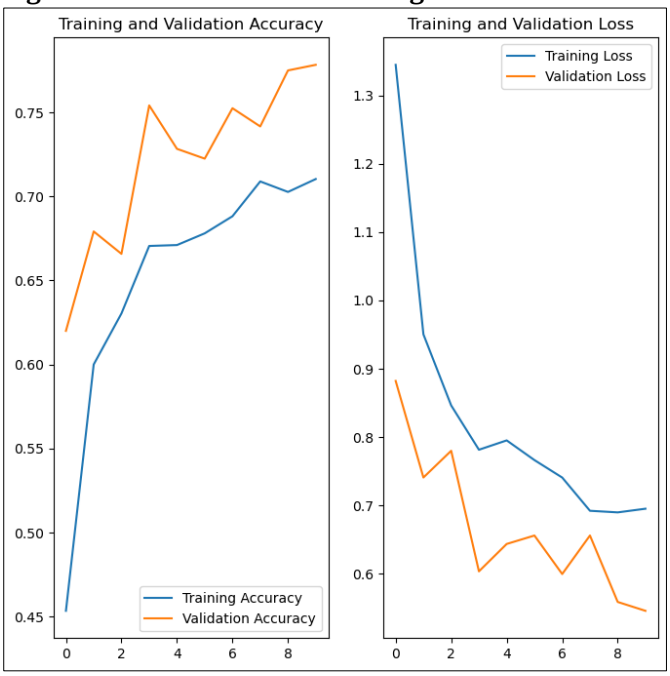
The VGG16 model achieves good training and validation performance. The training loss decreased consistently over the epochs, indicating that the model was effectively learning from the training data and reducing its training error. The validation loss also decreased steadily over the epochs, suggesting that the model was generalizing well to unseen data and minimizing overfitting (***Figure 8***).

With regard to accuracy, the training accuracy increased gradually and reached a reasonably high value of 0.71 , indicating that the model was learning to classify the training examples correctly. The validation accuracy also increased steadily and achieved a relatively high value of 0.78, demonstrating the model's ability to generalize well and classify new images

accurately. However, there is a possibility for further improvement by training of more epochs or adjusting the model's hyperparameters.

Overall, the new model based on VGG16 performs reasonably well on the classification of the images. It effectively used the pre-trained weights of the base VGG16 layers to extract useful features from images, which are then utilized by the new dense layers for classification. The training curves demonstrate that the model is learning and improving over time, achieving lower loss and higher accuracy values as the training progresses (***Figure 8***).

Figure 8: VGG16 Model Training and Validation Curves



3.5.3. Model evaluation

These results indicate that the VGG16 model achieved a test loss of approximately 0.533 and a test accuracy of about 77.5percent.

The loss value represents the difference between the predicted and actual labels of the test data. The test loss of 0.533 suggests that the model has relatively low prediction errors on the test data. The accuracy value represents the proportion of correctly classified samples out of the total test samples. The accuracy of 0.775 indicates that the model classified approximately 77.5percent of the test data correctly.

Overall, these evaluation results suggest that the VGG16 model performed well on the test data as it achieved a relatively low loss and a satisfactory accuracy, indicating that it can effectively

classify the images in the test set. This model has a better performance than the best model selected during the training process (model 2) which had a test accuracy of 66 percent.

3.6.Analysis of Variability in Results

Working with a relatively small dataset in the context of convolutional neural networks (CNNs), requires the analysis of the variability in the results to understand the stability and reliability of the model's performance. This analysis helps to assess how the model performs on unseen data. It also enables one to identify potential overfitting or underfitting issues, assess the impact of limited data variability, and the need for additional data collection or augmentation. (Renard et al., 2020).

Some of the approaches to analyze the variability in the results include the following;

Cross-validation which is a technique that partitions the dataset into multiple subsets or folds. The model is trained and evaluated multiple times, each time using a different fold as the validation set. By averaging the performance across multiple iterations, you can obtain a more reliable estimate of the model's performance and assess its variability.

Data augmentation techniques, such as random rotations and translations, can be applied to increase the variability of the training data. This increases the size of the dataset and assess the impact of data variability on the model's performance (Renard et al., 2020).

Learning curves provide an overview into the model's performance as the amount of training data increases. By plotting the training and validation performance metrics against the number of training examples, one can identify any potential issues related to dataset size and variability.

Bootstrapping is a resampling technique that involves drawing random samples with replacement from the original dataset. Training the model on multiple bootstrapped samples and evaluating the performance, can help with estimating the variability of the model's performance.

4.0.Statistical/Machine Learning

4.1.Introduction

This report analyses the Kvasir A Multi-Class Image Dataset .The dataset used in implementing statistical learning is the Kvasir-v2-features dataset of 9.3MB. The R caret package is used to implement the various tasks in the report

4.2.Construction of the features dataset with 1186 features

The Kvasir dataset consists of 4,000 images, categorized into 8 classes, with 500 images per class. The 8 classes include different anatomical landmarks, pathological findings, and endoscopic procedures related to gastrointestinal diseases(Pogorelov et al., 2017).

The anatomical landmarks are recognizable features in the gastrointestinal tract essential for navigation and act as reference points for pathology location. The landmarks in the dataset include the Z-line, pylorus, and cecum.

Pathological Findings are abnormal features within the gastrointestinal tract that indicate ongoing disease or precursors to conditions such as cancer. These findings in the dataset include esophagitis, polyps, and ulcerative colitis.

Endoscopic Procedures includes images related to the removal of polyps, specifically "dyed and lifted polyp" and "dyed resection margins".

All these specific features are important in multimedia research in areas such as automatic detection, classification and localization of endoscopic pathological findings in an image captured the GI tract. These formed the target variable that will help in the classification.

The paper used classification using global features (GF) where several image features for classification were extracted using the Lire open source software. The extracted features include; JCD, Tamura, ColorLayout, Edge Histogram, Auto Color Correlogram and Pyramid Histrogram of Oriented Gradients. For the 6 GF model, all extracted features were combined, resulting in the 1186 feature vector(Pogorelov et al., 2017).

For each image there were JCD-168, Tamura-18, ColorLayout-33, EdgeHistogram-80, AutoColorCorrelogram-256, PHOG-630 and each of these made up a feature, totalling up to 1185. The target which is one of the 8 classes formed the 1186th feature.

4.3. Construction of a 1186 feature vector using R for one of these feature files, i.e. for one image.

To create the feature vector, the first file in the first folder was used. The features in the file were extracted (JCD, Tamura, ColorLayout, EdgeHistogram, AutoColorCorrelogram, and PHOG) and a count was made of the number of valued under each feature. These values were used to create a vector list containing a repeat of each feature name according to the number of feature values under it plus the number of the vector under the specific feature. A data frame with columns equal to the length of the vector was constructed, with the feature vector names as the column names and the feature values from the file were recorded as a row under the column names. An index column was added to create a 1186 vector since the feature values were 1185 features.

The feature vector was created following the steps in R studio as explained below;

Defining the directory path and getting list of folders and files: The path to the directory where the data files are stored was initialized. The 'list.dirs()' function is then used to get a list of all folders in the specified directory. The 'list.files()' function is then used to get a list of all files in the first directory from the list of folders.

Reading the file and splitting the content: The 'readLines' function was used to read the content of the first file in the list of files, then the 'str_split()' function from the stringr package was then used to split the file content by line breaks (each line of the file is treated as a separate string).

Extracting feature names and counting feature values: An empty list called 'features_values_count' was created to store the results of the following loop. The loop went over each line of the file and each line was split by the semi-colon ":", resulting in two parts: the feature name and the feature values. The feature values were then split by a comma "," to get a list of individual values. The number of feature values was then counted and stored in the empty list with the feature name as the list key.

Creating the feature vector: An empty list 'feature_vectors' was created to store the results of the following loop. The loop went over each feature name in the features_values_count, and or each feature, a sequence of numbers from 1 to the count of the feature values is created and appended to the feature name, creating a vector of feature names with repetition corresponding to the number of feature values for each feature. These vectors were then combined into one long vector 'combined_vector' and the names were removed to create feature_vector1.

Creating a vector of feature values: This part of the code is very similar to steps 1-3, but instead of counting feature values, the actual values were extracted and stored in a list 'numbers_list'. The list of values was then converted to a one-dimensional array 'numbers_array'.

Creating a data frame with feature names and values: A data frame 'df' was then created from the 'numbers_array', with each row corresponding to a file and each column corresponding to a feature. The column names of the data frame are set to the feature names from 'feature_vector1'. An index column was added to the data frame and the data frame columns were then reordered so that the index column is the first column.

4.4. Using R code to construct a dataset of 1186 feature vectors for all the feature/image files.

To construct a dataset of 1186 feature vectors for all these feature/image files, r code was written to iterate through the folders and files within a specified directory, read the file contents, extract numeric values, and organize them into a final data frame. The resulting data frame contained the extracted values, along with columns indicating the source file and corresponding target or class. For those code the target column was added as the additional feature to facilitate the machine learning task of classification.

Below is the explanation of the steps taken in r code;

Define the path to the directory: The variable 'path_to_directory' was set to the path where the files were located.

Get the list of folders: The 'list.dirs()' function was used to retrieve a list of all folders within the specified directory.

Initialize an empty data frame: An empty data frame called 'df' was created to store the results.

Loop over the folders: An iteration over each folder in the folders list was done.

Get the list of files: The 'list.files()' function was then used to obtain a list of files within the current folder.

Loop over the files: An iteration over each file in the files list was also done.

Read the file: The content of the file was then read using the readLines() function, and the lines were stored in the variable lines.

Extract and process numbers: An empty list called numbers_list was initialized to store the extracted numbers. It then iterated over each line in the lines variable and split each line by the colon into two parts using strsplit(). The second part, which contained the numbers, was split by commas using strsplit()and the resulting numbers were converted to numeric format using as.numeric() and added to the numbers_list.

Convert the list to a one-dimensional array: The numbers_list was then converted to a one-dimensional array using unlist() and stored in the variable numbers_array.

Check the number of values: The length of numbers_array was compared with the length of feature_vector1 to ensure they had the same number of values. If they were not equal, an error message was printed, and the code skips to the next file using the next keyword.

Create a data frame for the file: A data frame called df_file was then created from the numbers_array, with a single row and columns corresponding to the feature_vector1 variable and the column names of df_file were set to feature_vector1.

Add a target column: A column called target was added to df_file, which contained the name of the current folder (obtained using basename(folder)).

Append the file data frame to the overall data frame: The df_file data frame was then appended to the main df data frame using rbind().

Add an index column: A new column called index was thereafter added to df, which contained a sequence of numbers from 1 to the number of rows in df.

Reorder the columns: Lastly, the columns of df were rearranged so that the index and target columns appear first, followed by the columns specified in feature_vector1.

4.5.Steps to reproduce the '6 GF Random Forrest' results

Information from report from the Kvasir: A Multi-Class Image-Dataset for Computer Aided Gastrointestinal Disease Detection, the parameters used to develop the 6 GF random forest model are as follows: 6 global features: JCD, Tamura, Color Layout, Edge Histogram, Auto Color Correlogram, feature vector size of size 1186, evaluation metrics: Precision, Recall,

Specificity, Accuracy, MCC (Matthews Correlation Coefficient), F1 Score, the Random forest parameters are not explicitly mentioned, the dataset used for training contains 250 images per class, with two equally-sized subsets for training and testing, Two-folded cross-validation, with the training and testing subsets switched, and the average performance calculated(Pogorelov et al., 2017).

To replicate these metrics, we used a random forest model, 2-fold cross validation and utilised all the 6 global features in the constructed 1186 features dataset constructed in the previous se ction. In order to asses whether we have reproduced the results, we shall use the same evaluat ion metrics; Precision, Recall, Specificity, Accuracy, and F1 Score. Tuning parameter 'mtry' w as held constant at a value of 34.42383 which was

Model building

We started by importing the required libraries including caret, random forest, ML metrics for the evaluation metrics, ml bench and the caTools library, which provides the sample.split function.

The data was then split into train and test sets using the sample.split function. The split ratio was set to 0.8, indicating that 80 percent of the data will be used for training. The index column, which was the first column in both the train and test data frames, was then removed using the subset operator as it is not required in the modelling process.

Defining target and predictors for train and test sets: Separate data frames (x_train, y_train, x_test, y_test) for the predictors and target variables for the training and testing sets were then created.

Building the random forest model: The training control parameters were defined using the trainControl function. It specified a repeated cross-validation method with 2 folds and 2 repeats, and , summary function set to ‘multiClassSummary. Other options, such as verbosity, class probabilities, selection function, and saving predictions, were set. The evaluation metric was set to 'Accuracy' using the metric variable.

Setting the mtry parameter: The mtry parameter was set to the square root of the number of predictors in the x_train data frame.

Creating the tuning grid: The tuning grid was created using the expand.grid function, specifying the mtry parameter.

Training the random forest model: The random forest model was trained using the train function. It used the rf method, the training data (train), the evaluation metric ('Accuracy'), and the tuning grid, and the training control parameters defined earlier.

Printing the model: Finally, the trained random forest model (rf_base) was printed to view the results. ***Table 4*** presents the results of the baseline model 'rf_base'.

Base Model Performance Against the 6GF Random Forest Model

The results in *Table 4* indicate that the Rf_base model has a higher mean precision compared to the 6 GF Random Forrest model. This indicates that Rf_base has a higher proportion of correctly predicted positive observations among all observations predicted as positive. Similarly, the Rf_base model has a higher mean recall compared to the 6 GF Random Forrest model. This implies that Rf_base has a higher proportion of correctly predicted positive observations among all actual positive observations.

In addition, the Rf_base model has a slightly higher mean specificity compared to the 6 GF Random Forrest model, implying that Rf_base has a higher proportion of correctly predicted negative observations among all actual negative observations. The Rf_base model also has a higher mean F1 score compared to the 6 GF Random Forrest model. The F1 score considers both precision and recall, providing a measure of overall model performance.

However, the 6 GF Random Forrest model has a higher mean accuracy of 0.93 compared to Rf_base (0.87) , suggesting that the 6 GF Random Forrest model has a higher overall proportion of correctly classified instances.

Overall, the Rf_base model outperforms the 6 GF Random Forrest model in terms of mean precision, mean recall, mean F1 score, and mean specificity. However, the 6 GF Random Forrest model has a higher mean accuracy, suggesting it has a better overall classification performance. The results can be improved by tuning some of the parameters including mtry.

Table 4: Evaluation Metrics for the Base Random Forest Model and 6 GF Random Forest Model

	Mean Precision	Mean Balanced Accuracy	Mean Recall	Mean Specificity	Mean F1 Score
Rf_base	0.7633119	0.8650833	0.7639248	0.9662417	0.7610126
6 GF Random Forrest	0.732	0.933	0.732	0.962	0.727

Source: (Pogorelov et al., 2017)

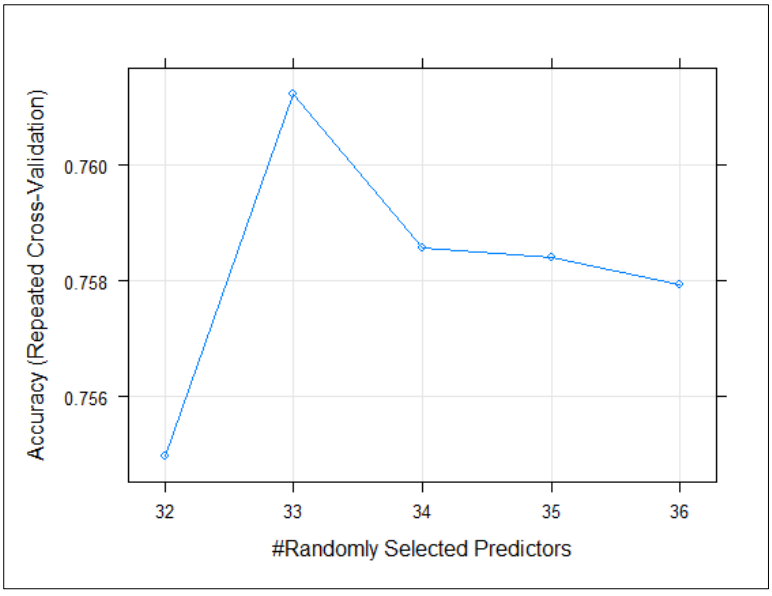
4.6.Tuned Base Random Forest Model

To improve the results, the mtry parameter was tuned using the expand grid function to select the mtry parameter value that provides the best results, with the range of mtry set to 32-36. The **cross fold validation parameter was retained at 2 folds with no repeats.**

Based on the results from the expand grid to obtain the best value for the parameter mtry. A value of 33 was selected as it had the highest accuracy of 76 percent (

Figure 9).

Figure 9: Accuracy at Different Mtry levels



Model Performance

Precision is the ratio of correctly predicted positive observations to the total predicted positives, implying that a higher value is the better. In this case, "Rf_tuned" (0.7613249) had a higher precision than "6 GF Random Forest" (0.732).

In terms of accuracy, the 6 GF Random Forest model (0.933) outperformed the Rf_tuned (0.86) model implying that the 6GF model has better accuracy. It is also noted that the base model has a higher accuracy than the tuned model, therefore tuning the model did not help to

improve accuracy. With more parameters tuned in a different random forest model and a larger number of mtry parameter cosidered , a performance could have been better.

In terms of recall, the Rf_tuned model (0.7615701) had a higher recall than the 6 GF Random Forest model (0.732). Similarly, the Rf_tuned model slightly outperforms (0.9658973) the 6 GF Random Forest (0.962) in terms of specificity.

Lastly, the F1 score for the Rf_tuned model (0.7585728) was higher than that of the 6 GF Random Forest model (0.727).

Similar to the base moel, the Rf_tuned model performs better in terms of Precision, Recall, Specificity, and F1 score, whereas the 6 GF Random Forest only outperforms it in accuracy.

Evaluation using rf_base

Model evaluation was used on the base model which was a better model than the tuned model as it had higher accuracy. The accuracy of the base model on the test set was 0.88 which is still lower than the 6GF model (**Table *5***). Therefore, if Accuracy is more important, you might choose the 6 GF Random Forest model, otherwise, the Rf_base model might be the better choice if the other metrics are more important.

Table 5: Evaluation Metrics for the Tuned Random Forest Model and 6 GF Random Forest Model

	Mean Precision	**Mean Balanced Accuracy**	**Mean Recall**	**Mean Specificity**	**Mean F1 Score**
Rf_tuned_validation	0.7613249	0.863733	0.7615701	0.9658973	0.7585728
6 GF Random Forrest	0.732	0.933	0.732	0.962	0.727
Rf_Base_test	0.7949653	0.88	0.7954123	0.97	0.79

5.0.References

Brownlee, J. (2019, April 16). How Do Convolutional Layers Work in Deep Learning Neural Networks? *MachineLearningMastery.Com*. https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/

CS231n. (2023). *CS231n Convolutional Neural Networks for Visual Recognition*. https://cs231n.github.io/convolutional-networks/

Jessica Powers. (2022). *What Is Transfer Learning? A Guide for Deep Learning | Built In*. https://builtin.com/data-science/transfer-learning

Mittal, S., & . V. (2021). A Survey of Accelerator Architectures for 3D Convolution Neural Networks. *Journal of Systems Architecture*, *115*. https://doi.org/10.1016/j.sysarc.2021.102041

Pogorelov, K., Randel, K. R., Griwodz, C., Eskeland, S. L., De Lange, T., Johansen, D., Spampinato, C., Dang-Nguyen, D.-T., Lux, M., Schmidt, P. T., Riegler, M., & Halvorsen, P. (2017). KVASIR: A Multi-Class Image Dataset for Computer Aided Gastrointestinal Disease Detection. *Proceedings of the 8th ACM on Multimedia Systems Conference*, 164–169. https://doi.org/10.1145/3083187.3083212

Renard, F., Guedria, S., Palma, N. D., & Vuillerme, N. (2020). Variability and reproducibility in deep learning for medical image segmentation. *Scientific Reports*, *10*(1), Article 1. https://doi.org/10.1038/s41598-020-69920-0

runhani. (2017, June 19). *Answer to "Intuitive understanding of 1D, 2D, and 3D convolutions in convolutional neural networks."* Stack Overflow. https://stackoverflow.com/a/44628011

Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data*, *6*(1), 60. https://doi.org/10.1186/s40537-019-0197-0

The MathWorks Inc. (2023). *What Is a Convolutional Neural Network? | 3 things you need to know*. https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html

Wikipedia. (2023). Transfer learning. In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Transfer_learning&oldid=1151630235

www.keras.io. (2023). *Keras documentation: VGG16 and VGG19*. https://keras.io/api/applications/vgg/

Yamashita, R., Nishio, M., Do, R. K. G., & Togashi, K. (2018). Convolutional neural networks: An overview and application in radiology. *Insights into Imaging*, *9*(4), Article 4. https://doi.org/10.1007/s13244-018-0639-9