BACHELOR'S DEGREE THESIS

Degree in Statistics

Title: Classification of medical images with convolutional networks

Author: Shengnan Li

Advisor: Ferran Reverter Comes

Department: Department of Genetics, Microbiology and Statistics

Academic year: July 2024

UNIVERSITAT DE BARCELONA

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Abstract

This study explores using Convolutional Neural Networks (CNN) to predict microsatellite instability (MSI) and stability (MSS) from histology images in gastrointestinal cancer.

A deep learning model was developed with Keras and TensorFlow in R, applying advanced techniques to histology images.

The results show that deep CNN architectures effectively predict MSI and MSS, providing clinicians with a reliable tool to identify the microsatellite stability of tumor tissues.

Keywords: Deep learning, CNN, medical image classification, MSI, MSS, gastrointestinal cancer, Keras, TensorFlow.

Este trabajo explora el uso de Redes Neuronales Convolucionales (CNN) para predecir la inestabilidad (MSI) y estabilidad microsatelital (MSS) a partir de imágenes histológicas en el cáncer gastrointestinal.

Se desarrolló un modelo de aprendizaje profundo con Keras y TensorFlow en R, aplicando técnicas avanzadas a las imágenes histológicas.

Los resultados muestran que las arquitecturas profundas de CNN predicen eficazmente MSI y MSS, proporcionando a los clínicos una herramienta confiable para identificar la estabilidad microsatelital de los tejidos tumorales.

Palabras Claves: Deep learning, CNN, Clasificación de imágenes médicas, MSI, MSS, Cáncer gastrointestinal, Keras, TensorFlow.

Classification AMS

- 62-07 Data analysis
- 62M45 Neural nets and related approaches
- 62P10 Applications to biology and medical sciences
- 68T05 Learning and adaptive systems [See also 68Q32, 91E40]

Acknowledgment

First and foremost, I would like to express my sincere gratitude to my advisor, Ferran Reverter Comes, for his invaluable support and guidance throughout the thesis creation process. His expertise and encouragement were instrumental in completing this work.

Additionally, I am deeply thankful to my family and friends for their unwavering support and encouragement, which have been a constant source of motivation.

Secondly, I would like to extend my special thanks to my best friend Xiaoxiang Liu for accompanying me through every struggling day, providing me with strength. Her companionship enabled me to persevere until the end.

And also I sincerely thank the FME servicio TIC team for their dedicated assistance in ensuring the normal operation of the server. I am deeply grateful for your professional support and timely and effective responses. It is precisely because of your vigorous assistance that can ensure the stable and efficient operation of the system.

Finally, I would like to thank my university. These four years have flown by, and I am grateful for the guidance of every teacher and the platform the school has provided. These four years of university have given me invaluable knowledge and beautiful memories.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Background and motivation

With the rapid development of artificial intelligence and machine learning, the application of deep learning in medical image classification has become an important research field[7]. Especially in gastrointestinal cancer diagnosis, the prediction of type of cancer like microsatellite instability (MSI) and stability (MSS) by analyzing biopsy images is of great significance for improving diagnostic accuracy and efficiency[4]. This study aims to use convolutional neural networks (CNN) to classify histological images of gastrointestinal cancer to predict MSI and MSS, thereby providing clinicians with a reliable tool to identify microsatellite stability of tumor tissue.

Hypothesis

The core hypothesis of this study is that by using a convolutional neural network, combined with appropriate data preprocessing and model optimization techniques, we can accurately classify histological images of gastrointestinal cancer and effectively distinguish between MSI and MSS.

Objective

The main objective of this study was to explore and implement the application of convolutional neural networks in medical image classification, especially for biopsy image classification in gastrointestinal cancer diagnosis[6]. Specific objectives include:

1. Acquire CNN knowledge: A comprehensive understanding of the theoretical concepts and practical applications of convolutional neural networks.

2. Implementing CNN in R: Learn and apply techniques for implementing convolutional neural networks using R and Keras libraries.

3. Training CNN to analyze images: Develop and train a CNN model capable of analyzing biopsy images to identify patterns and features associated with cancer.

4. Data Analysis: Implement a complete data analysis process, including data preprocessing, model training, validation, and testing.

5. Report writing: Document the research process, methods, results and conclusions in detail.

Structure

The structure of this paper is following:

1. Introduction: Introduction of research background, motivation, hypothesis, and research purpose.

2. Methodology: The specific description of data and preprocessing, model construction and training, model verification and testing are introduced.

3. Theoretical framework: An overview of the relevant theoretical background, including the basic concepts of artificial intelligence, machine learning, deep learning, and convolutional neural networks.

4. Practice framework: Describes the practical steps in the research in detail, including the specific implementation process of data description, data preprocessing, model construction, training, validation and testing.

5. Results and discussion: Analyze the training and validation results of the model, and discuss the research findings and possible directions for improvement.

6. Conclusions: The main findings of the study are summarized, the limitations of the study are discussed, and suggestions for future research are presented.

Through these sections, this paper will comprehensively present the research process, results and conclusions of using convolutional neural networks to classify histological images of gastrointestinal cancer.

2 Methodology

Data preprocessing

This study used a dataset of histological images from the Zendoo[5] database, specifically a subset with image fragments from patients with gastrointestinal cancer. The images were color normalized using the Macenko method and stored in a size of 224 x 224 pixels. The data sets were divided into training sets, validation sets, and test sets, and were allocated in a ratio of 2:1:1.

Specific data preprocessing steps include:

1. Image renaming : Rename all pictures according to MSI and MSS to MSI.1.jpg, MSI.2.jpg...MSS.1.jpg, MSS.2.jpg,... etc. for easy management and processing.

2. Data segmentation : The image data set is divided into training set, validation set and test set, which are stored in different folders. The specific allocation is: training set 2,000, verification set 1,000, test set 1,000, a total of 4,000 images.

3. Data normalization : The value of each pixel is re-scaled to the range [0,1] for better processing and calculation by the neural network.

Model construction and training

This study mainly constructs and trains five different convolutional neural network models, detailed as follows:

1. Model 1 : Build a model that includes multiple convolutional layers, pooling layers, and fully connected layers. The input size is 224 x 224 x 3 and the output is binary (MSI and MSS). The cross entropy loss function and RMSprop optimizer were used for compilation and training, with 50 training rounds.

2. Model 2: In order to solve the overfitting problem, the size of the input image is adjusted to 64 x 64 x 3.

3. Model 3 : Introduce data augmentation and Dropout to increase the amount of training data and improve the generalization ability of the model by randomly transforming and discarding some neurons.

4. Model 4: Add a batch normalization layer after each convolutional layer to speed up the training process and improve model stability.

5. Model 5: A pre-trained VGG16 network is introduced for feature extraction and fine-tuning. The implementation includes two methods: fast feature extraction without data augmentation (Model 5.1), and feature extraction combined with data augmentation (Model 5.2). At the same time, fine-tuning (Model 5.3) and end-to-end training (Model 5.4) were performed.

Training and verification

All models were trained using a training data generator and a validation data generator, setting the appropriate batch size and training rounds. The performance of the model is evaluated by plotting training loss and validation loss curves, and adjustments are made as needed to reduce overfitting.

In addition, the model was trained on cat and dog datasets to provide a reference for model validation. For details, please refer to Appendix 8.1.

Data analysis and results report

Record and analyze model training and validation metrics through a complete data analysis process, including data preprocessing, model training, validation, and testing. And also to evaluate the level of the data representation by doing PCA, t-SNE and UMAP.

Write a detailed research report providing implementation details and performance evaluation of the model.

3 Theoretical Framework

This section outlines the theoretical framework, which includes the key theories and concepts that form the basis of the research. It provides a comprehensive overview of the relevant literature and theoretical perspectives that inform the study.

3.1 Artificial Intelligence, Machine Learning, Deep Learning

First of all, before understanding convolutional neural networks, we need to understand some other concepts, the relationship between artificial intelligence, machine learning and deep learning. Figure 3.1 below can clearly see the relationship between them.

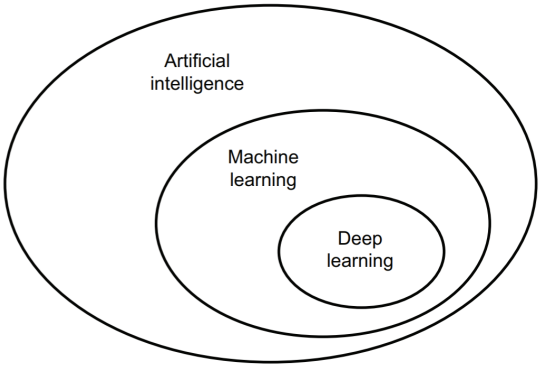


Figure 3.1: Artificial intelligence, Machine learning, Deep learning

The main role of artificial intelligence is to try to automate intellectual tasks normally performed by humans. Therefore, artificial intelligence is a comprehensive field that includes not only machine learning and deep learning, but also more methods that do not involve learning. For example, chess rules written by programmers. The way machine learning works is to give the machine data and answers, and the machine itself summarizes the rules.

Deep learning is a branch of machine learning that emphasizes learning from successive layers. In deep learning, 'deep' does not refer to the deeper level of understanding achieved using this approach, but rather to a series of successive layers of representation. The number of layers contained in the data model is called the depth of the model.

In deep learning, these layers representations are almost always learned through models called neural networks. Although her name is reminiscent of neurobiology, some of her core concepts in deep learning are inspired in part by people's understanding of the brain. The two are related, but not the same. The diagram (Figure 3.2) shows their similarities, which we will learn more about in the following sections.

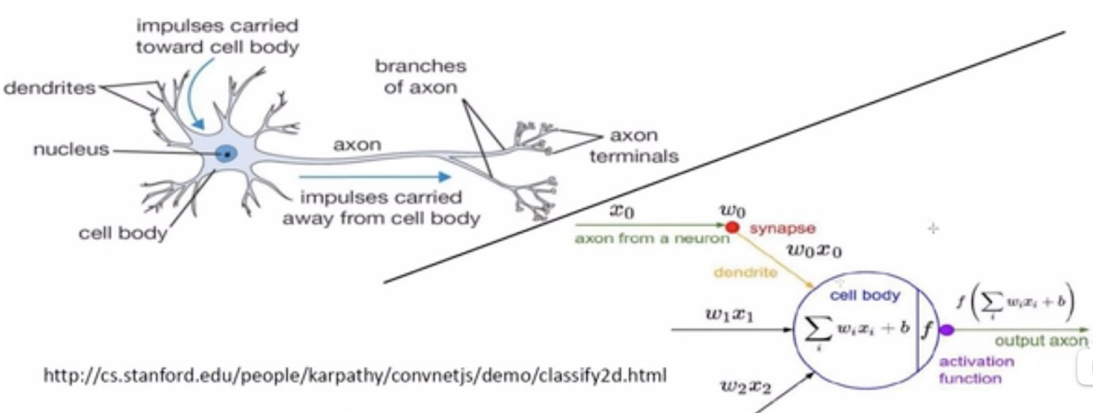


Figure 3.2: Biological Neural Networks vs Neural Networks

3.2 The basics of neural networks

This section will introduce some basic knowledge of neural networks, loss function, activation function, gradient descent.

3.2.1 Activation Function

Sigmoid activation function: $s = \sigma(w^T x + b) = \sigma(z) = \frac{1}{1+e^{-z}}$, which enables the neural network to perform nonlinear transformations after the calculation by the weight parameters w.

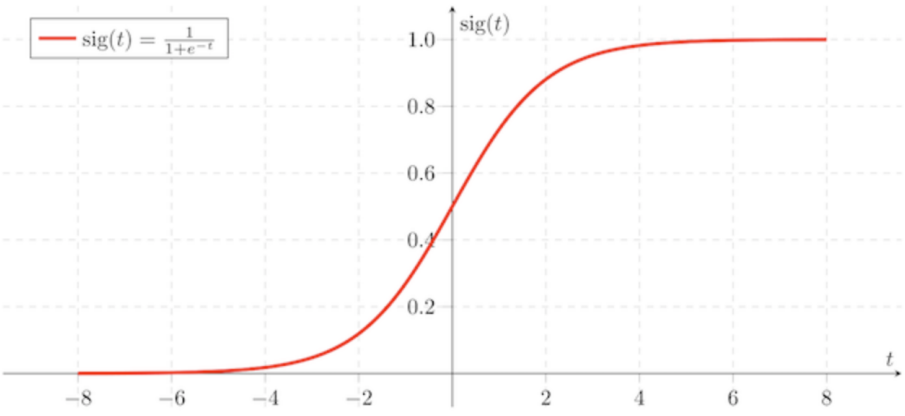


Figure 3.3: Sigmoid activation function

- If the result of z is very large, then the result of s is close to 1.
- If the result of z is small or a very large negative number, then the result of s is close to 0.

Relu activation function

The Relu(rectified linear unit) function is the most efficient in the gradient propagation, with all

negative numbers on the feature plot becoming 0 and the rest unchanged.

$$RELU(x) = max(x, 0)$$

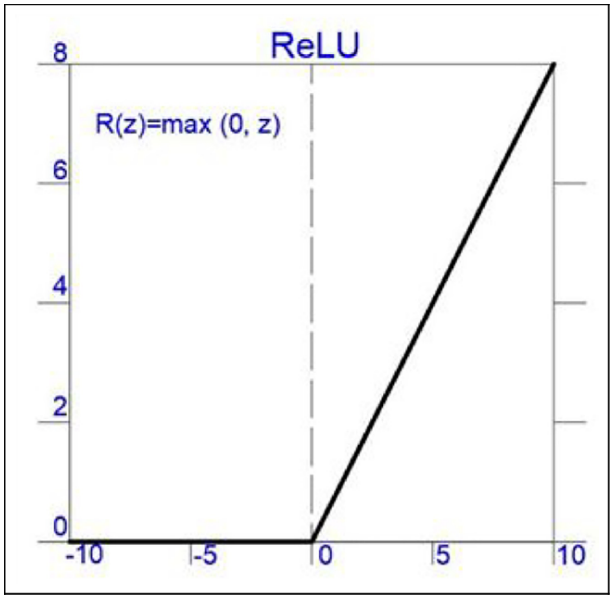


Figure 3.4: ReLU activation function

3.2.2 Loss function

The loss function is used to measure the discrepancy between the predicted result and the true value. The simplest definition of the loss function is the mean squared error loss:

$$L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

3.2.3 Gradient descent

The purpose of the gradient descent algorithm is to minimize the value of the loss function. The gradient of the function indicates the steepest growth direction of the function. In the direction of the gradient, the function grows faster. If you go in the negative direction of the gradient, the function value will naturally decrease the fastest. The training goal of the model is to find the appropriate weight(w) and bias(b) to minimize the lost function value.

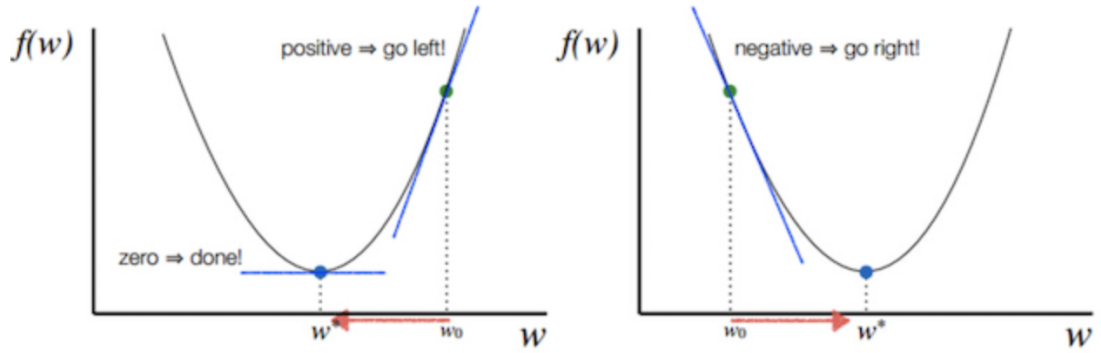


Figure 3.5: Gradient descent

3.3 Deep neural networks

Deep learning algorithms can be imagined as a multi-level information distillation operation, in which the information is continuously filtered and gradually purified. Deep learning can learn from data because it has two basic characteristics: first, the way of increasing layer by layer, the higher the number of layers, the more complex the expression; The second is that these intermediate increments can be learned together, and each layer update must meet the representation needs of the previous and the next layer.

We will see how it works in three pictures. Are the three main steps of deep learning: 1. Input data points. 2. Sample of expected output. 3. Measure how good the algorithm is.

The specification of what the layer does with its input data is stored in the layer's weights, which are actually a set of numbers. In other words, the transformations implemented by the layers are parameterized by their weights (Figure 3.6). Learning here means finding a set of values for the weights of all layers of the network so that the network can correctly map the sample input to the associated target.

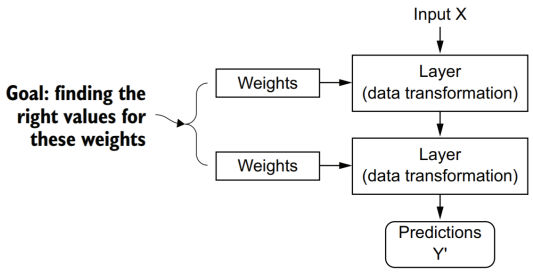


Figure 3.6: A neural network is parameterized by its weights

However, deep neural networks may contain tens of millions of parameters, and modifying

the value of one parameter will affect all other parameters, so in order to control the output of the neural network, need to be able to measure the difference between this value and the expected value, which is where the loss function is derived. The loss function takes the network prediction and the real target as parameters, and calculates the distance score to obtain the performance of the network in this specific example (Figure 3.7). This process is called the forward propagation algorithm.

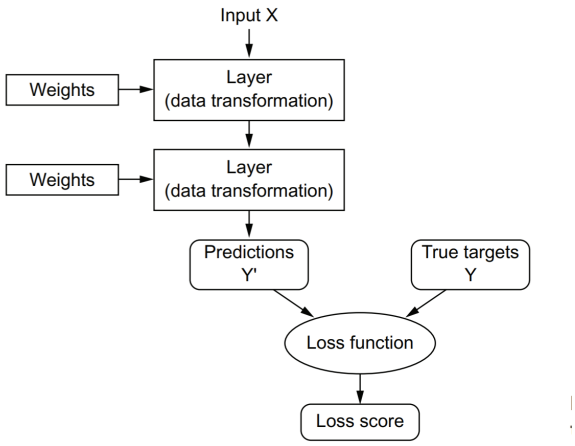


Figure 3.7: A loss function measures the quality of the network's output

The basic trick of deep learning is to use the loss value as a feedback signal to fine-tune the weight and thus reduce the loss value. This process is done by the optimizer by implementing a back-propagation algorithm. Gradient descent is the core learning mechanism. The model starts with randomly initialized weights, calculates the weights by back-propagation, and gradually adjusts the weights according to the errors between the network output and the expected output.(Figure 3.8)

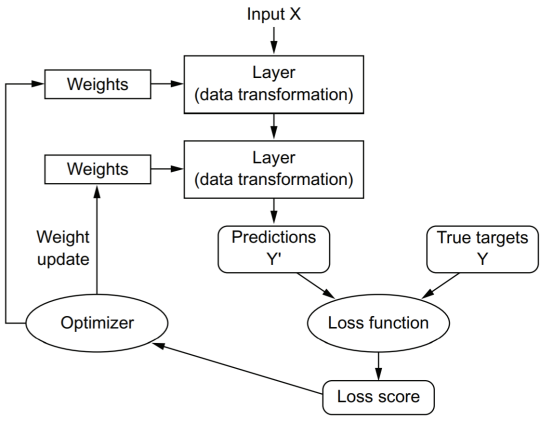


Figure 3.8: The loss score is used as a feedback signal to adjust weights

These are the three main steps.

3.4 Convolutional Neural Networks(CNN)

From the previous section, we have learned about deep learning and neural networks. Starting from this section, we will focus on introducing convolutional neural networks that will be used in this study[3].

3.4.1 Why we use CNN

In the field of computer vision, what is usually done is to identify the target image with a machine program instead of the human eye. In fact, both neural networks and convolutional neural networks have existed in the last century, but in recent years, with the enhancement of computer computing power and the increase of training data, neural network algorithms have become popular again, so convolutional neural networks are also popular.

If a grayscale image of size 28x28, it is a single channel image, and each pixel has only one value, then there are 784 features in total. If it is color then are three channels (RGB channels), which will be 28x28x3, 2352 values. If we had 10 neurons, we would have 23,520 weight parameters. If the picture is larger, let us say 1000x1000x3, there will be 30 million weight parameters. Such parameter size is computationally large, and it is difficult to achieve better results, so there is the popularity of convolutional neural networks.

3.4.2 Principles of CNN

Here will introduce the structure of the neural network and the principle of convolution to understand the pooling and calculation process[2].

3.4.2.1 The composition of CNN

Convolutional neural networks consist of one or more convolution layers, pooling layers, and fully connected layers. Compared to the structure of deep learning, convolutional neural networks can give better results in terms of images and so on. This model can also be trained using a back propagation algorithm. Compared with other deep neural networks, convolutional neural networks need to consider fewer parameters. The following image(Figure 3.9) shows the overall structure of a convolutional neural network:

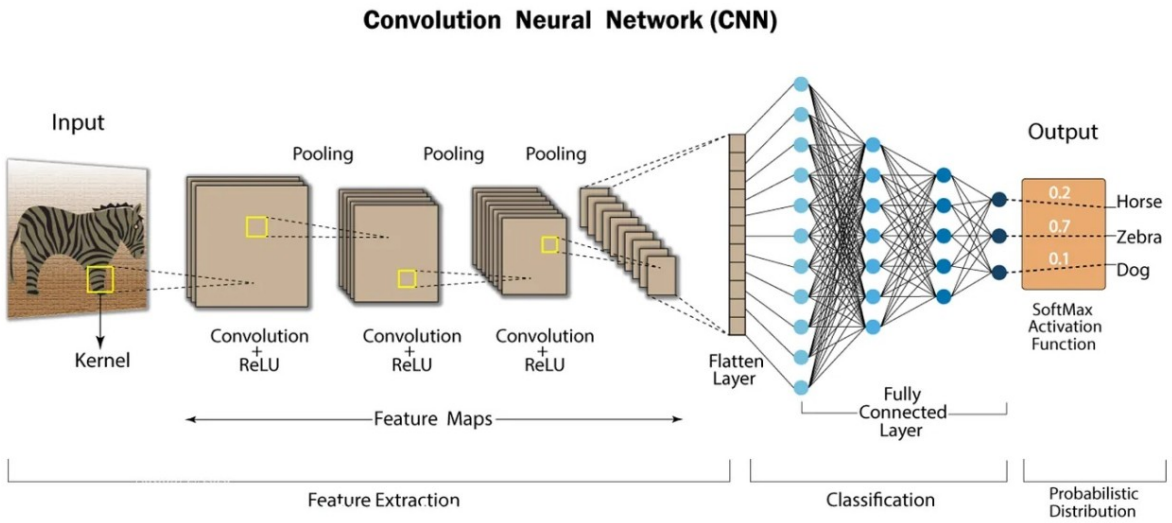


Figure 3.9: An example of CNN structure

It contains several main structures:
1. Convolution layer
2. Pooling layer
3. Fully connected layer

3.4.2.2 Convolution layer

The purpose of convolution operation is to extract different features of input and get eigenvalues. Some convolution layers may only be able to extract some low-level features such as edges, lines and corners, and more layers of the network can iteratively extract more complex features from low-level features.

Convolution operates on tensors in three dimensions, called feature graphs, with height axes, width axes, and depth axes. For an RGB image with a depth axis size of 3, the convolution operation extracts small blocks from its input feature map and applies the same transformation to all the small blocks to generate the output feature map. This output feature map is still a three-dimensional tensor with width and height. Depth can be arbitrary, because the output depth is a parameter of the layer, and the different channels in the depth axis no longer represent a specific color, as RGB inputs do; Instead they represent filters. The filter can not only be designed manually, but also the value of the filter can be used as a parameter and learned through back propagation. The output picture is a diagram of the filter's response to the input.

For example, a 28x28x1 feature graph with an output size of 28x28x32 would mean that he calculates 32 filters on his input. Each of these 32 output channels contains a 26x26 grid value, which is the response graph.

Convolution is defined by two key parameters:
1. The size of the block extracted from the output is called the convolution kernel and is usually 3×3 or 5×5 of odd size.
2. The depth of the output feature map is the number of filters computed by the convolution, usually 32,64,128.

Convolution generates an operation result by using the convolution kernel. As shown in the Figure 3.10, a 5x5 single-channel image is obtained by using a 3x3 size convolution kernel operation to obtain a 3x3 size result.

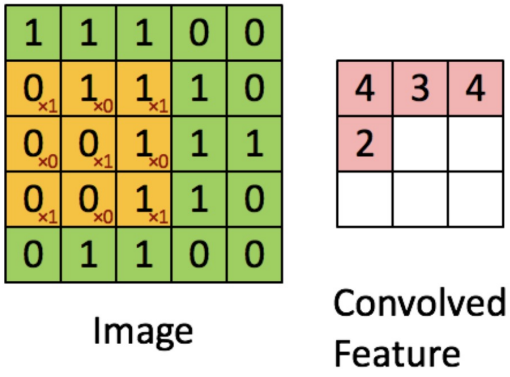


Figure 3.10: Convolution

If N is the size of the image and F is the size of the convolution kernel, then $N - F + 1 = 5 - 3 + 1 = 3$.

If we change the convolution kernel or add many layers of convolution, the image may end up being 1x1, which is not what we want. And for the edge pixels in the original image, the value is calculated only once, for the middle pixels will be calculated many times with the filter, which will lead to the loss of edge information. So we introduce zero padding here.

3.4.2.3 Padding

In order for the output feature map to have the same spatial dimension as the input, here can use padding. Padding consists of adding the appropriate number of rows and columns on each side of the input feature map so that the center of the corresponding window can be fitted around each input block. For 3x3 windows, can add one column on the upper, lower, left and right sides, and for 5x5 windows, can add two columns.

The value of added pixels is 0, the number of added layers is denoted as p, if added 1 layer p=1. The reason for adding 0 is that 0 does not affect the final result in the weight calculation product and operation, and avoids adding additional disturbing information to the picture. The step size of the move is one pixel, and the outer layer of 0 is added, so we can calculate with

the formula: $5 + 2 * p - 3 + 1 = 5$

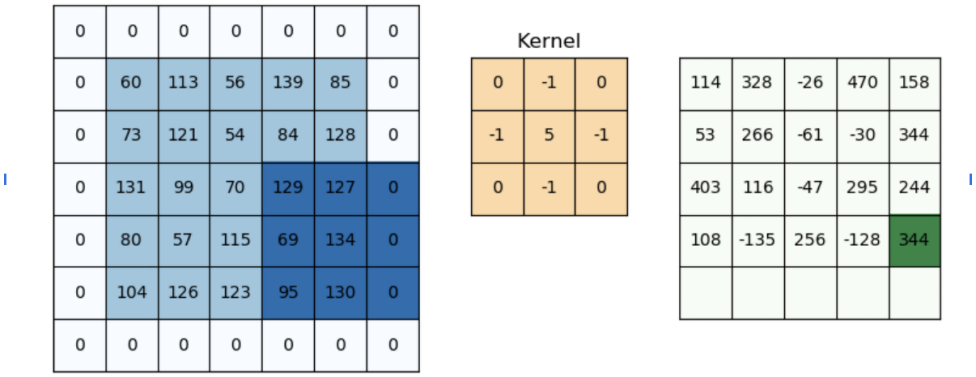


Figure 3.11: Padding

If the padding size is 2, then the picture size is 7, which is larger than the previous picture.

There are two types of padding, the Same mode is more selective:
1. Valid: Not padding, that is, the final size is $(N - F + 1) * (N - F + 1)$
2. Same: The output size is the same as the original size, then N becomes $N + 2P$, the final size is going to be $(N + 2P - F + 1) * (N + 2P - F + 1)$, then $N = N + 2P - F + 1$. If the output is to be equal, it must be filled with zeros of size $P = \frac{F-1}{2}$, so when we know the size of the convolution kernel, we can figure out the number of layers to be filled.

From the above formula, we can see that if F is not odd but even, the result of the final P is not an integer, resulting in uneven filling, so this is why the convolution kernel defaults to the odd latitude size. And the filter of odd latitude has a central value, which is convenient to indicate the position of the filter.

3.4.2.4 Stride

The stride is the number of steps kernel moves, usually one stride at a time. If the step size is 2, the result size will change.

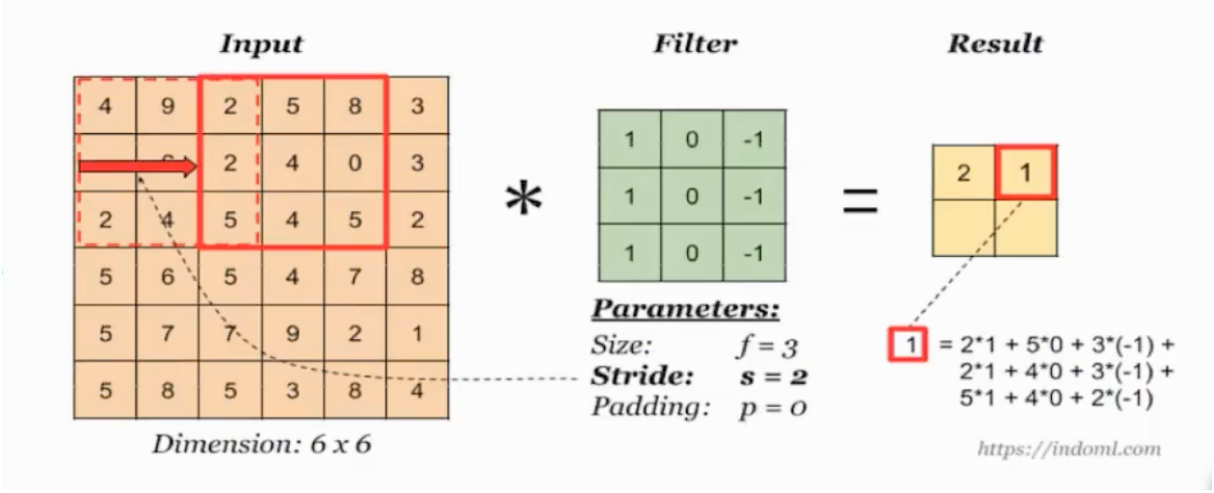


Figure 3.12: Stride

When the original step size was 1, then $N + 2P - F + 1 = 6 + 0 - 3 + 1 = 4$.

Now the step size is 2 to get a result, so it will be changed to $\frac{(N+2P-F)}{2} + 1 = 2.5$, the result is a decimal, integer down, is 2.

So, when the input picture size is N, the filter size is F, the step size is S, and the zero fill is P, the final formula is: $(\frac{(N+2P-F)}{S} + 1, \frac{(N+2P-F)}{S} + 1)$

3.4.2.5 Multichannel convolution

If the input layer consists of multiple channels, such as RGB three channels, the convolutional kernel must have the same number of channels. Each convolutional kernel channel is convolved with the corresponding channel in the input layer. The convolution results of each channel are combined to obtain a final feature map.

Therefore, if a multi-channel color picture has only one filter, the number of channels of the filter and the number of channels of the picture must be the same, and the parameter is $3 * 3 * 3 = 27$

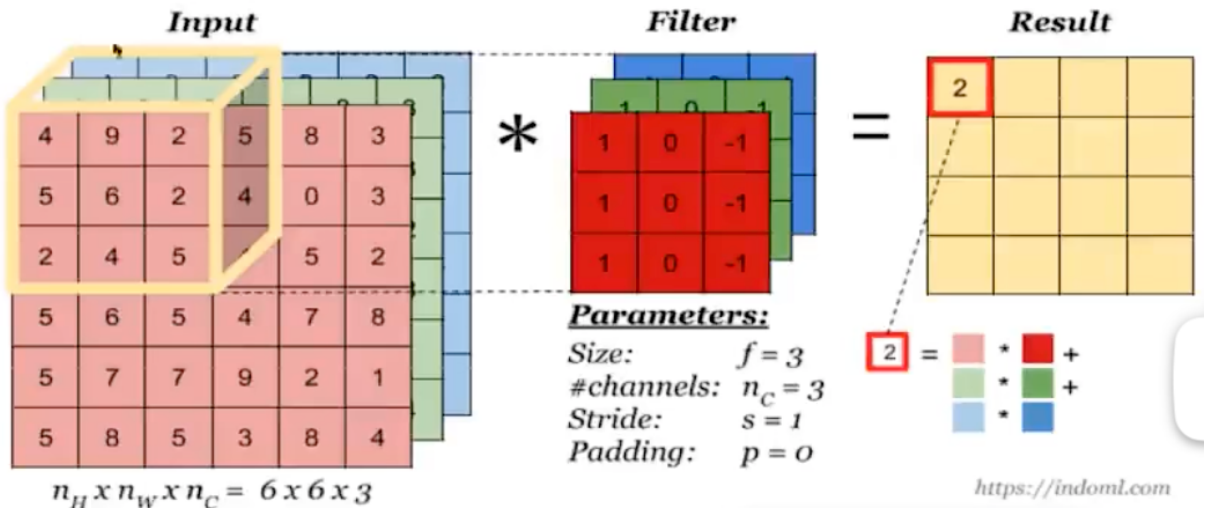


Figure 3.13: Multichannel convolution

3.4.2.6 Multiple convolution kernel

When there are multiple convolutional kernels, a variety of different features can be learned and a feature map containing multiple channels can be generated accordingly. For example, two filters are shown on the graph, so the output of two channels can be understood as multiple convolutional kernels or neurons. The size of the resulting feature map has no effect, but the number varies. The number of convolution kernel filters is equal to the number of feature graphs.

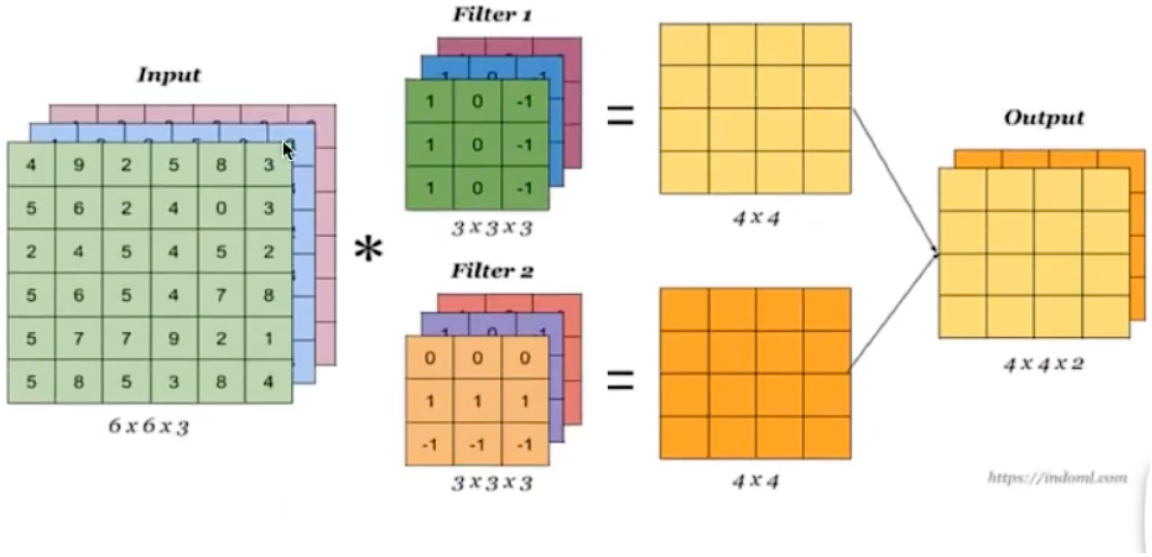


Figure 3.14: Multiple convolution kernel

3.4.2.7 Pooling layer

The pooling layer is mainly used to subsample the feature graphs learned by the convolution layer, which can be divided into two types.

1. Averaging pooling: Averaging pooling takes the average value of all values in the window as the output.
2. Max pooling: Max pooling extracts the window from the input feature map and outputs the maximum value of each channel. It is conceptually similar to convolution, except that instead of transforming local small pieces by a learned linear transformation, they are transformed by a hard-coded maximum-tensor operation. A big difference from convolution is that maximum pooling is usually done with 2x2 Windows and step size 2, in order to downsample the feature graph by a factor of 2.

The significance of this is to further enlarge the main features and ignore the deviation of a few pixels, thus reducing the input latitude of the subsequent network layer, reducing the training parameters, reducing the size of the model, improving the calculation speed, and avoiding overfitting.

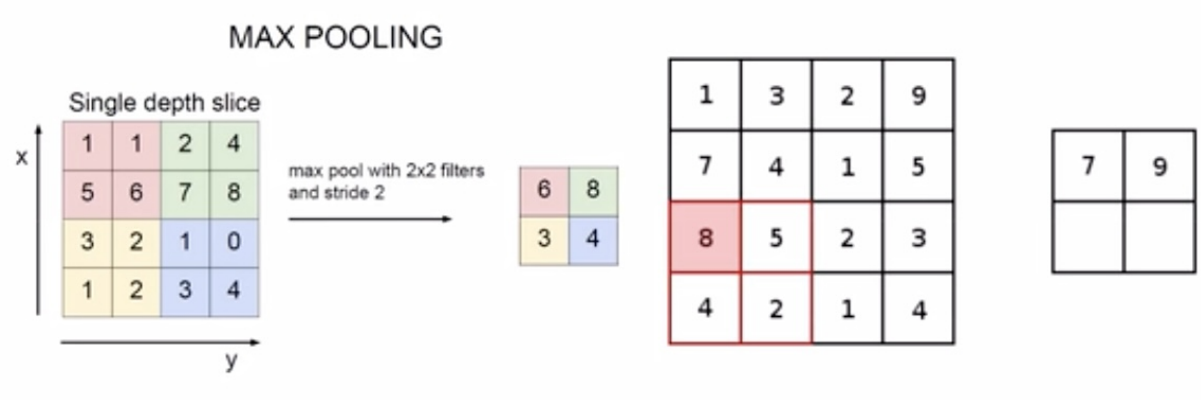


Figure 3.15: Max pooling

For the input image(Figure 3.15), using a parameter with a region of 2*2 and a stride of 2 to do the operation of maximizing.

Similarly, pooling also has a set of parameters f and s, f is the window size, s is the stride, p is padding, usually the default is $f = 2 * 2, s = 2$.

Pooling parameter features: no parameters, no need to learn, unlike convolution through gradient descent update.

3.4.2.8 Fully connected layer

Convolution layer, activation layer and pooling layer together can be regarded as the feature learning layer of CNN, that is, feature extraction layer, and the learned features are finally applied to the model tasks such as classification problems and regression problems.

First, the feature map is flattened, that is, reshaped into a 1*N vector to become a shallow neural network with n features. And then add one or more connected layers for model learning, like showed in following Figure 3.16.

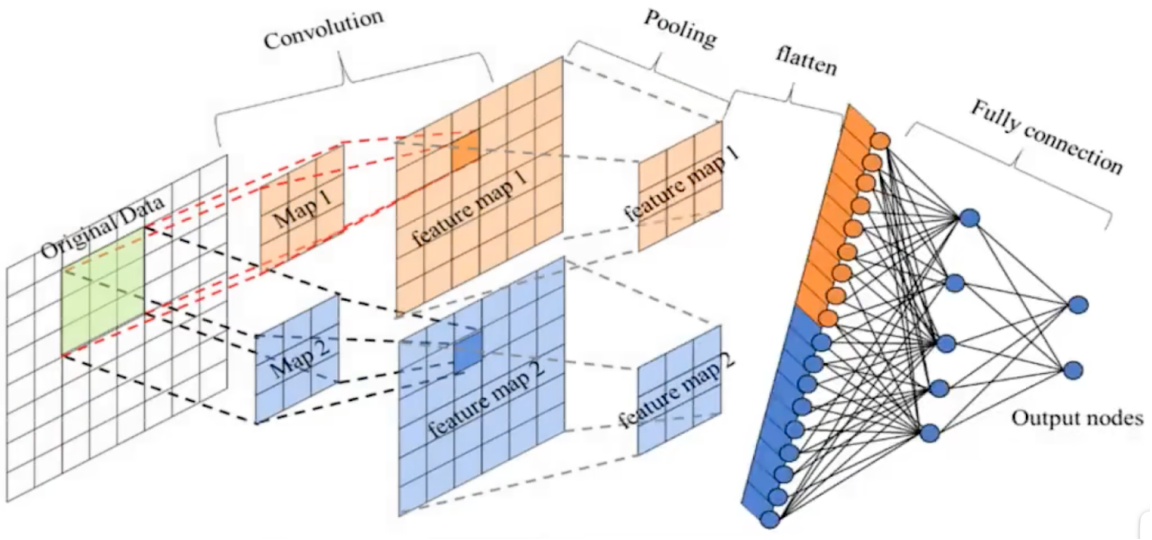


Figure 3.16: Fully connected layer

4 Practical Framework

This section presents the practical framework, detailing the methods and procedures used in the research. It includes a description of the research design, data description, and analytical methods employed to address the research questions.

4.1 Description of the data

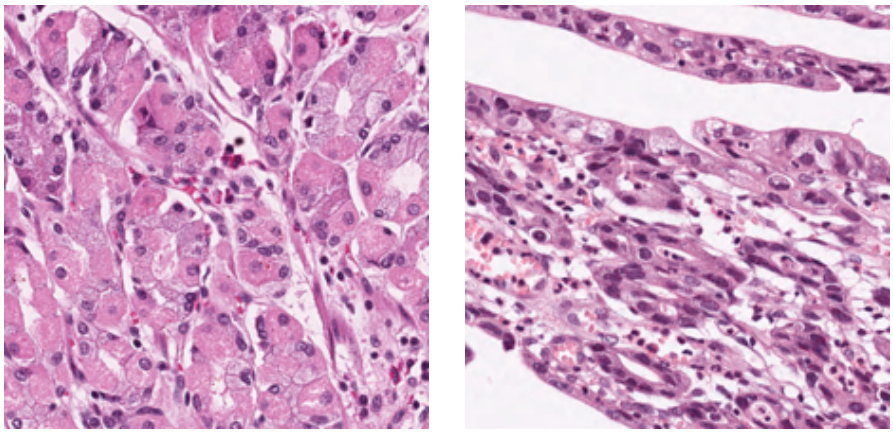
In this section, will introduce the source of these image data and the basic information of these images.

4.1.1 Origin of the data

This set of data is from the Zendoo database. This repository contains 411,890 unique image patches derived from histological images of colorectal cancer and gastric cancer patients

Because the size of the data is massive, two sets of images are selected as research objects in this study, names are STAD_TRAIN_MSS[1] and STAD_TRAIN_MSIMUT[2]. Are histological images for gastric (stomach) cancer.

These images are in size of 224 px x 224 px and color normalization with the Macenko method, in 3-color channel RGB format.



(a) MSI.1 (b) MSS.1

Figure 4.1: Images of MSI and MSS

4.1.2 Data preprocessing

First we will arrange the image as MSI.1.jpg,MSI.2.jpg... this format renames all images in the MSI folder so that all images have a more concise numbering. And create three folders, train,

[1]training images (70% of all patients) for gastric (stomach) cancer TCGA patients with MSS (microsatellite stable) tumors, 50285 unique image patches; FFPE samples

[2]training images (70% of all patients) for gastric (stomach) cancer TCGA patients with MSI (microsatellite instable) or highly mutated tumors, 50285 unique image patches; FFPE samples

validation, test. Divide the images into these three folders in a 2:1:1 ratio. Similarly, the MSS folder is also processed this way. See the table 1 below for specific assigned data sizes.

	Train	Validation	Test	Total
MSI	1,000	500	500	2,000
MSS	1,000	500	500	2,000
Total	2,000	1,000	1,000	4,000

Table 1: Data size 1

The value of each pixel ranges from 0 to 255. In order to make the neural network better compute, we re-scale its original data to the interval of [0,1].

4.2 Model 1

First, we build a model with multiple convolution layers, pooling layers, and fully connected layers. The input size is 224x224x3, the output shape of the fourth layer of maximum pooling is 10x10x128, and the flattened vector is processed by two fully connected layers before producing a final output value, which is activated by the Sigmoid function for binary classification. The total number of trainable parameters and trainable parameters in the final model is 7,012,289 (Figure 4.2).

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_7 (Conv2D)           (None, 218, 218, 32)      4736
 max_pooling2d_7 (MaxPooling2D)  (None, 109, 109, 32)  0
 conv2d_6 (Conv2D)           (None, 103, 103, 64)      100416
 max_pooling2d_6 (MaxPooling2D)  (None, 51, 51, 64)    0
 conv2d_5 (Conv2D)           (None, 47, 47, 128)       204928
 max_pooling2d_5 (MaxPooling2D)  (None, 23, 23, 128)   0
 conv2d_4 (Conv2D)           (None, 21, 21, 128)       147584
 max_pooling2d_4 (MaxPooling2D)  (None, 10, 10, 128)   0
 flatten_1 (Flatten)         (None, 12800)             0
 dense_3 (Dense)             (None, 512)               6554112
 dense_2 (Dense)             (None, 1)                 513
=================================================================
Total params: 7012289 (26.75 MB)
Trainable params: 7012289 (26.75 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 4.2: Summary of Model 1

Using a data generator, the re-scaled data can be used to generate training data of batch size 20 and also for validation data of batch size 20 to facilitate image preprocessing and batch loading, thereby improving training efficiency and reducing memory footprint.

After that, the model was compiled and binary cross-entropy was selected as the loss function, since the task was a binary classification problem, the images are going to be classified into

MSI and MSS. The optimizer is chosen to use the RMSprop optimizer and set the learning rate to 1^{-4} to ensure that the model could eventually be brought into the model during the training process and the convergence was stable enough. Accuracy is selected as an evaluation metric to measure the model's performance on the validation set.

Finally, we bring the picture into the model and train the model using the training data generator and the validation data generator, with the epoch set to 100 steps, that is, 100 batches per epoch. The epoch number is set to 50, meaning that the model will be trained 50 times on the entire data set.

The results obtained are shown in the Figure4.3 below.

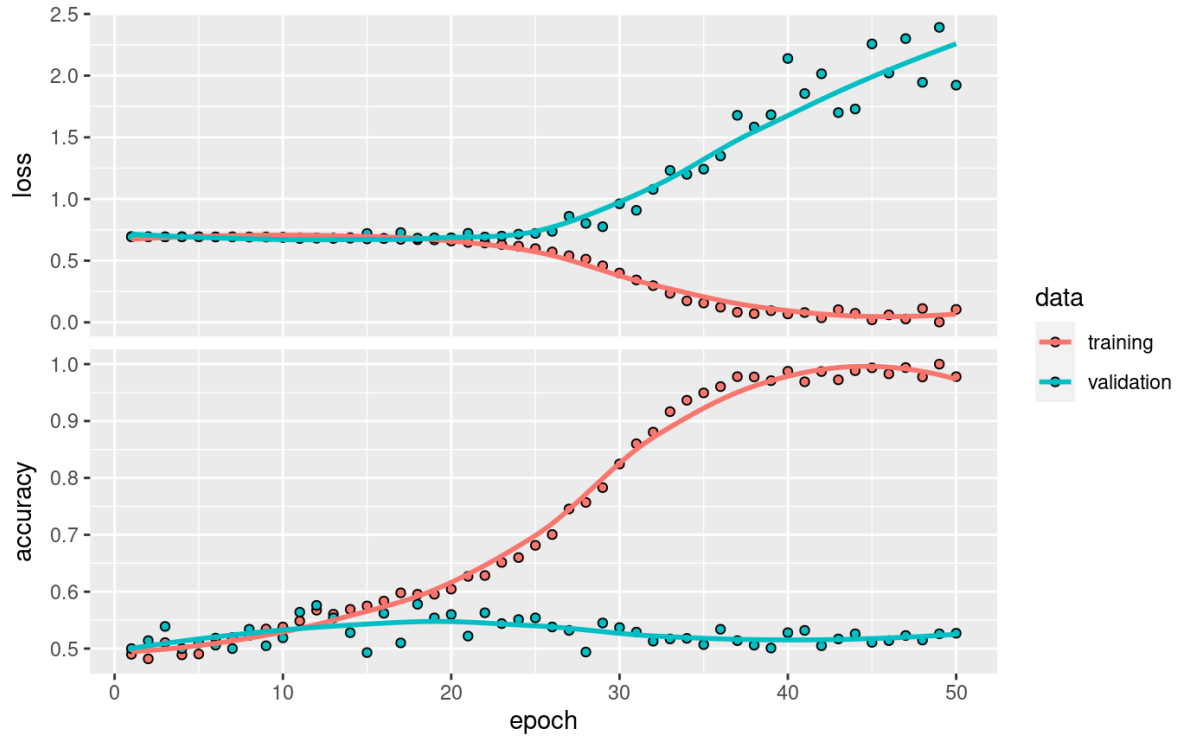


Figure 4.3: Training indicators and validation indicators of Model 1

We plot the loss and accuracy of the model during training. This graph reflects the characteristics of overfitting. The training accuracy increases over time until it reaches nearly 100%, while the validation accuracy stays at 50%, as if nothing has been learned. The validation loss reaches a minimum around round 25 and then stops, while the training loss keeps decreasing linearly until it approaches 0.

4.3 Model 2 64X64

Due to the overfitting, we will do further processing to adjust the data so that it has a better accurate value of validation.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_7 (Conv2D)           (None, 58, 58, 32)        4736
 max_pooling2d_7 (MaxPooling2D)  (None, 29, 29, 32)    0
 conv2d_6 (Conv2D)           (None, 25, 25, 64)        51264
 max_pooling2d_6 (MaxPooling2D)  (None, 12, 12, 64)    0
 conv2d_5 (Conv2D)           (None, 8, 8, 128)         204928
 max_pooling2d_5 (MaxPooling2D)  (None, 4, 4, 128)     0
 conv2d_4 (Conv2D)           (None, 2, 2, 128)         147584
 max_pooling2d_4 (MaxPooling2D)  (None, 1, 1, 128)     0
 flatten_1 (Flatten)         (None, 128)               0
 dense_3 (Dense)             (None, 512)               66048
 dense_2 (Dense)             (None, 1)                 513
=================================================================
Total params: 475073 (1.81 MB)
Trainable params: 475073 (1.81 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 4.4: Summary of Model 2.1

Therefore, we adjusted the size of the image in model 2, and adjusted the input size to 64x64x3 in the image generator, and finally generated the following Figure 4.4. After the fourth maxpoling layer, the image has become a picture with the size of 1x1x128, and the size of 1*1 has lost too much information. To keep more information we delete the last convolution layer and pooling layer, and the final model is as follows (Figure 4.5), the final total number of parameters is reduced from 7,012,289 to 1,310,529.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 58, 58, 32)        4736
 max_pooling2d_6 (MaxPooling2D)  (None, 29, 29, 32)    0
 conv2d_5 (Conv2D)           (None, 25, 25, 64)        51264
 max_pooling2d_5 (MaxPooling2D)  (None, 12, 12, 64)    0
 conv2d_4 (Conv2D)           (None, 8, 8, 128)         204928
 max_pooling2d_4 (MaxPooling2D)  (None, 4, 4, 128)     0
 flatten_1 (Flatten)         (None, 2048)              0
 dense_3 (Dense)             (None, 512)               1049088
 dense_2 (Dense)             (None, 1)                 513
=================================================================
Total params: 1310529 (5.00 MB)
Trainable params: 1310529 (5.00 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 4.5: Summary of Model 2

The results obtained are shown in the figure below, which still has the characteristics of over-fitting. Compared with the previous model, there is little change, it can be said that there is

almost no change.

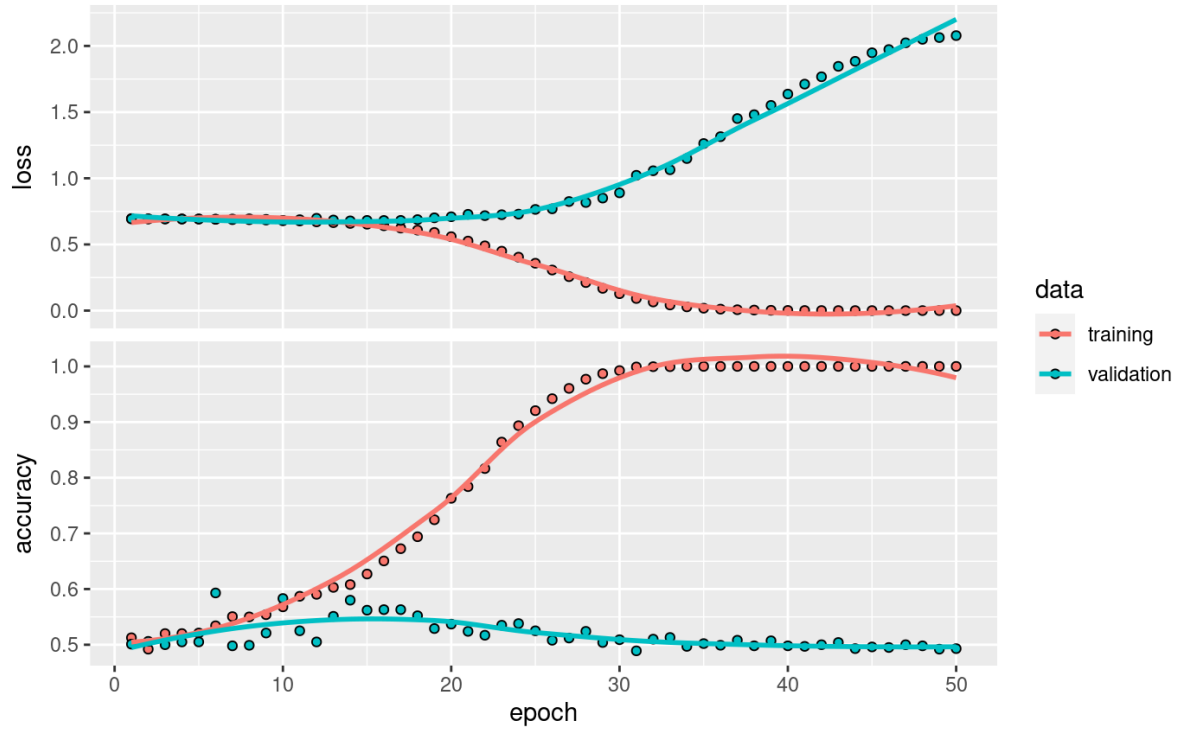


Figure 4.6: Training indicators and validation indicators of Model 2

4.4 Model 3 Data augmentation and Drop-out

As model 2 continues to exhibit the characteristics of overfitting, we introduce the concepts of data augmentation and dropout and continue to train the model.

Data augmentation uses the method of generating more training data from existing training samples, increasing the sample by a large number of random transformations, by randomly translating the picture vertically or horizontally, randomly cutting changes, randomly scaling the inside of the picture, randomly inverting the general picture in the horizontal direction, and filling in newly created pixels, etc., so as to produce a credible image.

Dropout is a regularization technique that selects a few neurons in each iteration of each layer not to use, but will be used in the next iteration, for example, only 50% of the neurons are trained each time, and the next iteration selects another 50% of the neurons for iteration. The goal is that in the model never sees the exact same picture twice while training.

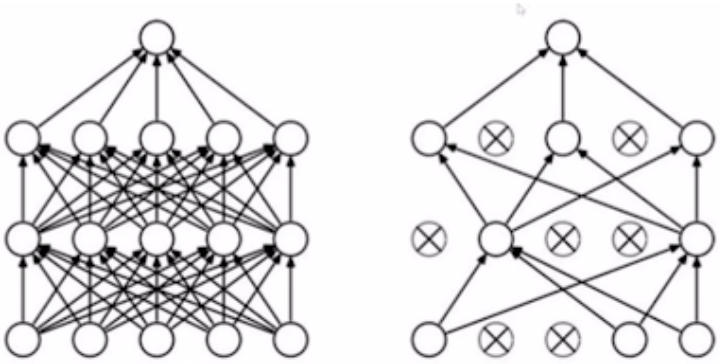


Figure 4.7: Dropout

This helps expose the model to more aspects of the data and better generalize.

After adding these two layers we can see the result of the following model, the total number of parameters has not changed.

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_17 (Conv2D)          (None, 58, 58, 32)        4736
 max_pooling2d_17 (MaxPooling2D) (None, 29, 29, 32)    0
 conv2d_16 (Conv2D)          (None, 25, 25, 64)        51264
 max_pooling2d_16 (MaxPooling2D) (None, 12, 12, 64)    0
 conv2d_15 (Conv2D)          (None, 8, 8, 128)         204928
 max_pooling2d_15 (MaxPooling2D) (None, 4, 4, 128)     0
 flatten_4 (Flatten)         (None, 2048)              0
 dropout (Dropout)           (None, 2048)              0
 dense_9 (Dense)             (None, 512)               1049088
 dense_8 (Dense)             (None, 1)                 513
=================================================================
Total params: 1310529 (5.00 MB)
Trainable params: 1310529 (5.00 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 4.8: Summary of Model 3

The results show that overfiting has some improvement.

With the increase of training rounds, the training loss gradually decreases, indicating that the performance of the model on the training data is gradually improved. The loss value gradually decreased from near 0.70 to near 0.68. The validation loss also decreased in the early stage of training, but fluctuated greatly in the middle and late stages of training, and the overall trend increased. The loss value gradually decreased from close to 0.70 to about 0.69 and then increased.

The training accuracy rate increased gradually with the increase of the number of training

rounds, from about 0.50 to close to 0.58, indicating that the classification ability of the model on the training data was gradually enhanced. The accuracy of validation increased in the early stage of training, but fluctuated greatly in the middle and late stages, and the overall trend declined. The accuracy rate increased from about 0.50 to about 0.55 and then decreased.

The results show that although the training loss and accuracy perform well, the model may be still have overfitting.

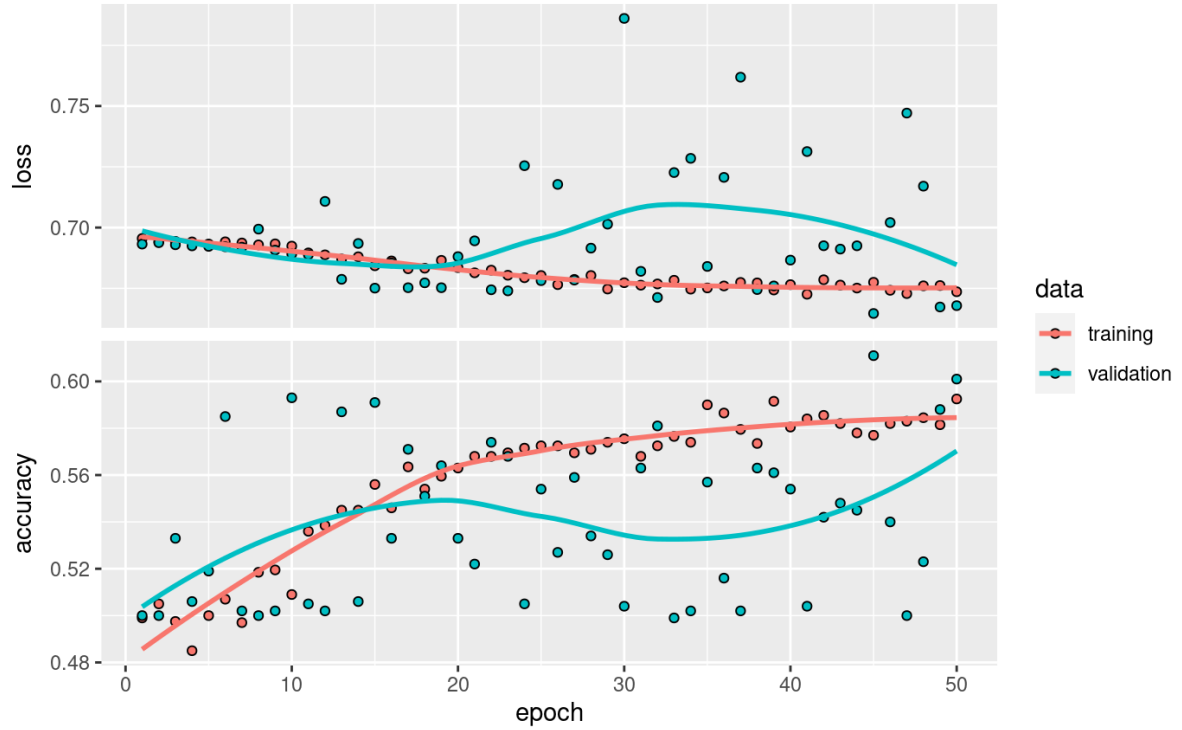


Figure 4.9: Training indicators and validation indicators of Model 3

4.5 Model 4 Normalization

In order to further solve overfiting, we added a batch normalization layer in model 4. After each convolutional layer, a batch normalization layer was added to speed up the training process and improve the model stability. These layers introduce additional parameters (128, 256, and 512) but the number of parameters is relatively small. The addition of batch normalized layers helps to speed up model training and avoid gradient disappearance or explosion problems.

```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   Trainable
=================================================================
 conv2d_27 (Conv2D)          (None, 58, 58, 32)        4736      Y
 batch_normalization_9 (BatchNormali  (None, 58, 58, 32)        128       Y
 zation)
 max_pooling2d_27 (MaxPooling2D)  (None, 29, 29, 32)        0         Y
 conv2d_26 (Conv2D)          (None, 25, 25, 64)        51264     Y
 batch_normalization_8 (BatchNormali  (None, 25, 25, 64)        256       Y
 zation)
 max_pooling2d_26 (MaxPooling2D)  (None, 12, 12, 64)        0         Y
 conv2d_25 (Conv2D)          (None, 8, 8, 128)         204928    Y
 batch_normalization_7 (BatchNormali  (None, 8, 8, 128)         512       Y
 zation)
 max_pooling2d_25 (MaxPooling2D)  (None, 4, 4, 128)         0         Y
 flatten_7 (Flatten)         (None, 2048)              0         Y
 dropout_3 (Dropout)         (None, 2048)              0         Y
 dense_15 (Dense)            (None, 512)               1049088   Y
 dense_14 (Dense)            (None, 1)                 513       Y
=================================================================
Total params: 1311425 (5.00 MB)
Trainable params: 1310977 (5.00 MB)
Non-trainable params: 448 (1.75 KB)
```

Figure 4.10: Summary of model 4

The plot shows how the model performed during training. Although the model performs well on the training data with an accuracy of about 0.6, it is unstable on the validation data with an upward trend in loss and a slight decline in accuracy.

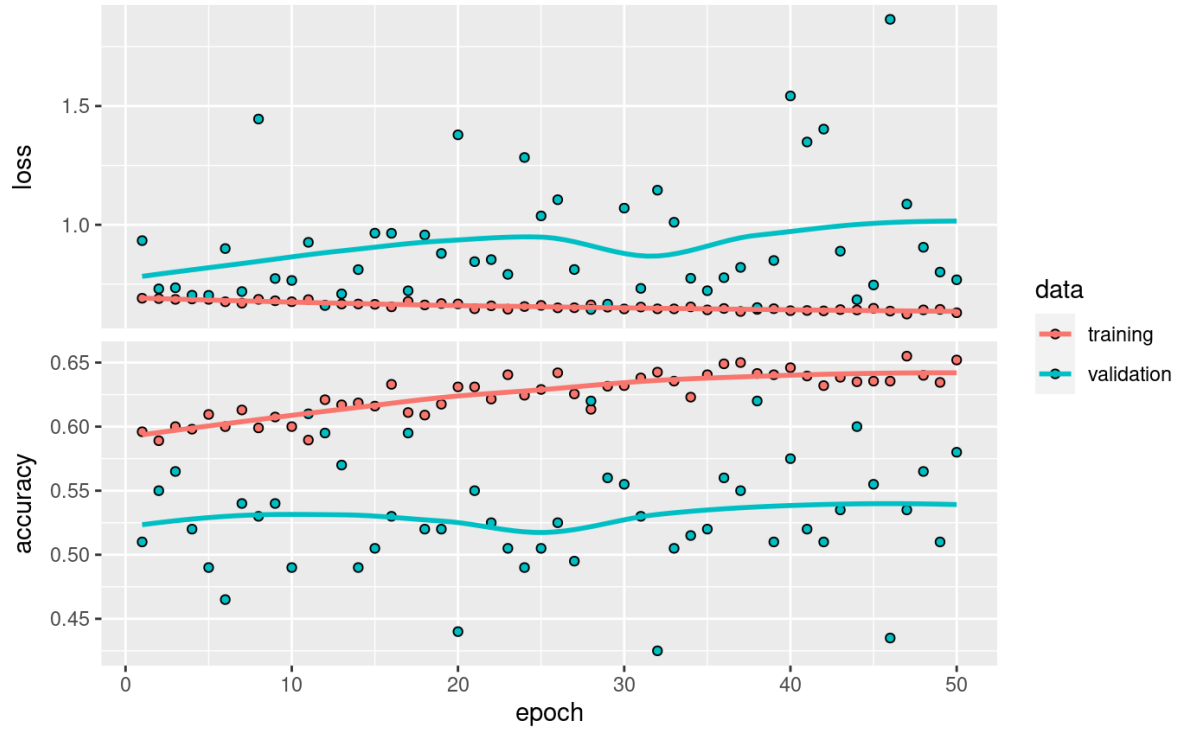


Figure 4.11: Training indicators and validation indicators of Model 4

4.6 Model 5 VGG16

Since this overfiting has been still exist, we decided to introduce the pre-trained network VGG16. Deep learning on small image datasets referencing some pre-trained networks is also a common and efficient approach. A pre-trained network is a saved network previously trained on a large dataset, usually on a large-scale image classification task. There are two ways to use a pre-trained network: feature extraction and fine-tuning.

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 64, 64, 3)]       0
 block1_conv1 (Conv2D)       (None, 64, 64, 64)        1792
 block1_conv2 (Conv2D)       (None, 64, 64, 64)        36928
 block1_pool (MaxPooling2D)  (None, 32, 32, 64)        0
 block2_conv1 (Conv2D)       (None, 32, 32, 128)       73856
 block2_conv2 (Conv2D)       (None, 32, 32, 128)       147584
 block2_pool (MaxPooling2D)  (None, 16, 16, 128)       0
 block3_conv1 (Conv2D)       (None, 16, 16, 256)       295168
 block3_conv2 (Conv2D)       (None, 16, 16, 256)       590080
 block3_conv3 (Conv2D)       (None, 16, 16, 256)       590080
 block3_pool (MaxPooling2D)  (None, 8, 8, 256)         0
 block4_conv1 (Conv2D)       (None, 8, 8, 512)         1180160
 block4_conv2 (Conv2D)       (None, 8, 8, 512)         2359808
 block4_conv3 (Conv2D)       (None, 8, 8, 512)         2359808
 block4_pool (MaxPooling2D)  (None, 4, 4, 512)         0
 block5_conv1 (Conv2D)       (None, 4, 4, 512)         2359808
 block5_conv2 (Conv2D)       (None, 4, 4, 512)         2359808
 block5_conv3 (Conv2D)       (None, 4, 4, 512)         2359808
 block5_pool (MaxPooling2D)  (None, 2, 2, 512)         0
=================================================================
Total params: 14714688 (56.13 MB)
Trainable params: 14714688 (56.13 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 4.12: Model of VGG16

4.6.1 Model 5.1 and Model 5.2 Feature extraction

Feature extraction involves extracting features from new samples using representations previously learned by the network. These functions are then run through a new classifier. As mentioned earlier, the network for image classification consists of two parts, starting with a series of pooled layers and convolutional layers, and ending with a full link layer classification. The first part is called the convolutional base of the model. By introducing the VGG16 model, we can see that the final feature map has the format 2x2x512, and we will add a dense link classifier at this layer. We're going to do this in two ways.

4.6.1.1 Model 5.1 Fast feature extraction without data augmentation

Run a convolution base on a data set, record its output in an array on disk, and then use this data as an independent, Input to a fully connected classifier. This scheme only needs to run the convolution base once for each input image, but does not allow the use of data augmentation.

As shown in the figure, the extracted feature maps currently have format 2*2*512, which is 2048 after sending them into the fully connected layer.

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 2, 2, 512)]       0
 flatten_9 (Flatten)         (None, 2048)              0
 dense_19 (Dense)            (None, 256)               524544
 dropout_5 (Dropout)         (None, 256)               0
 dense_18 (Dense)            (None, 1)                 257
=================================================================
Total params: 524801 (2.00 MB)
Trainable params: 524801 (2.00 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 4.13: Summary of Model 5.1

The final training results are shown in the figure below.

The training loss decreases significantly, indicating that the model's performance on the training data has gradually improved. The loss value gradually decreases from close to 25 to close to 0, indicating that the model has a good fitting effect on the training set. The validation loss also decreased significantly, and the model fitted well on the validation set.

The training accuracy rate increased from about 0.55 to close to 0.75, indicating that the classification ability of the model on the training data was gradually enhanced. The validation accuracy also gradually improved, from about 0.50 to nearly 0.60, which was slightly lower than the training accuracy, but also showed that the model had improved the classification ability on the validation data.

The chart shows well the model performs during training and validation. Both training loss and validation loss decreased significantly, and the accuracy rate increased significantly, indicating that the model had a good fitting effect on the training data and validation data, and no obvious overfitting phenomenon appeared.

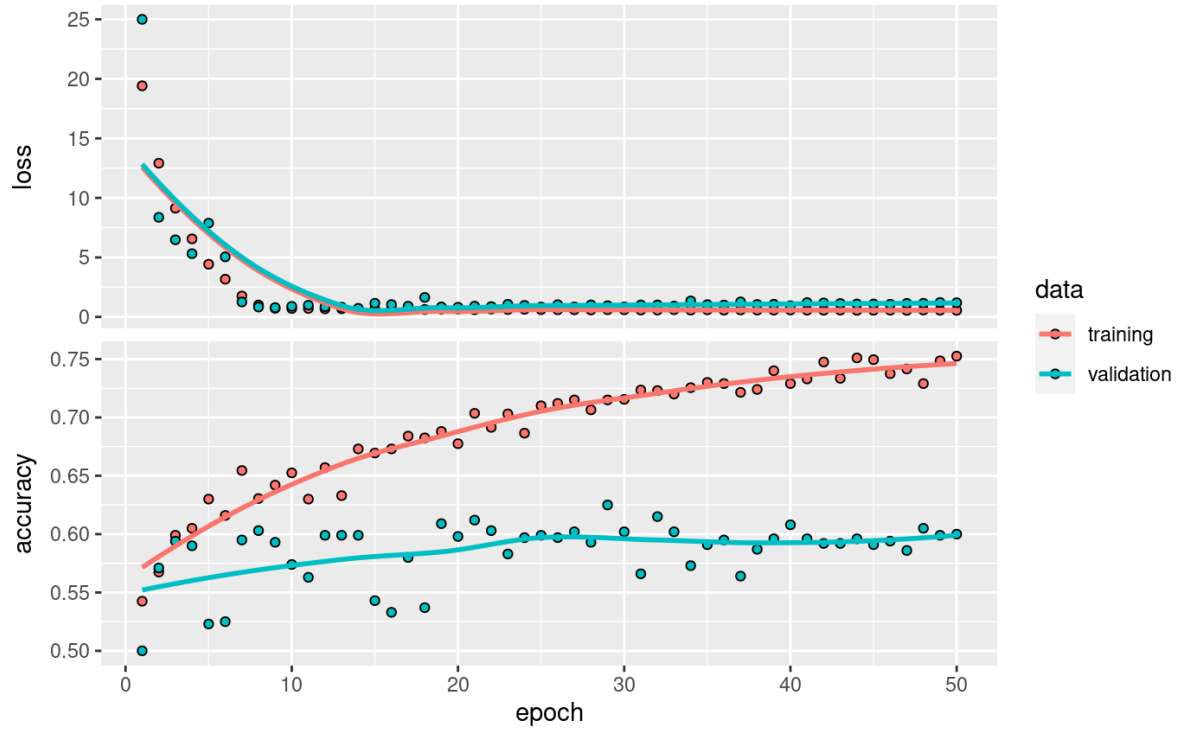


Figure 4.14: Training indicators and validation indicators of Model 5.1

4.6.1.2 Model5.2 Feature extraction with data augmentation

This model will add a full-link layer at the top and run the entire object end-to-end over the input data, which will allow data augmentation to be used because each input image will pass through the convolution base.

We can see that there are 14,714,688 parameters in the convolution base, which we will freeze before compiling and training the model, meaning that the weights will not be changed during training. In this model, it will train only the weights of the two fully linked layers added. The trainable parameters are 524,801.

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 vgg16 (Functional)          (None, 2, 2, 512)         14714688  
 flatten_10 (Flatten)        (None, 2048)              0         
 dense_21 (Dense)            (None, 256)               524544    
 dense_20 (Dense)            (None, 1)                 257       
=================================================================
Total params: 15239489 (58.13 MB)
Trainable params: 524801 (2.00 MB)
Non-trainable params: 14714688 (56.13 MB)
_________________________________________________________________
```

Figure 4.15: Summary of Model 5.2

The results of the drawing were observed again, and the accuracy training loss gradually decreased with the increase of the number of training rounds, indicating that the model's performance on the training data gradually improved. The loss value gradually decreased from close to 0.70 to close to 0.65. Validation loss also decreased gradually with the increase of training rounds, and leveled off at the later stage of training, with the loss value gradually decreasing from close to 0.70 to close to 0.60.

The training accuracy rate increased gradually with the increase of the number of training rounds, from about 0.54 to close to 0.63, and the verification accuracy rate also increased gradually with the increase of the number of training rounds, from about 0.55 to close to 0.61. Although it was slightly lower than the training accuracy rate, it also indicated that the model's classification ability on the verification data had been enhanced.

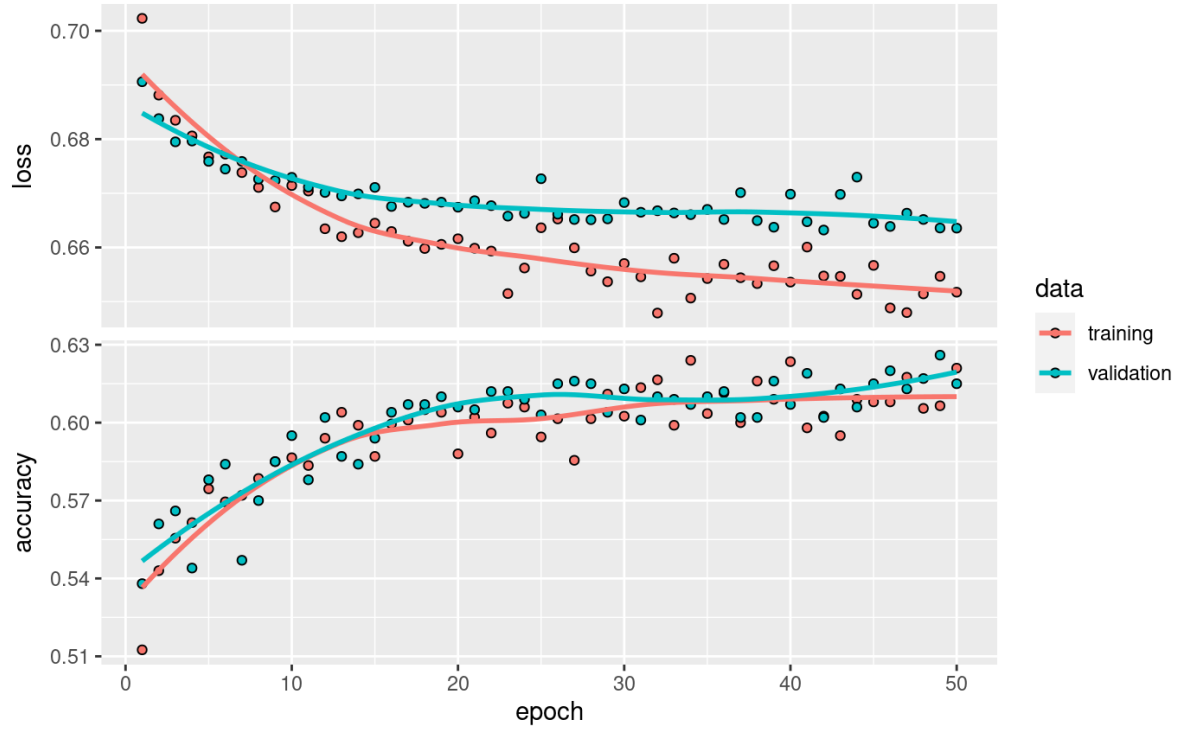


Figure 4.16: Training indicators and validation indicators of Model 5.2

4.6.2 Model 5.3 Fine tuning

Because the accuracy of 0.6 is still low, continue to adjust the model to achieve higher accuracy.

For the next model, we will fine-tune all the layers on block3_conv1, fine-tuning the lower layers will have a fast falling return, and the more parameters you train, the greater the risk of overfitting, so starting with the penultimate convolution block will be a good choice.

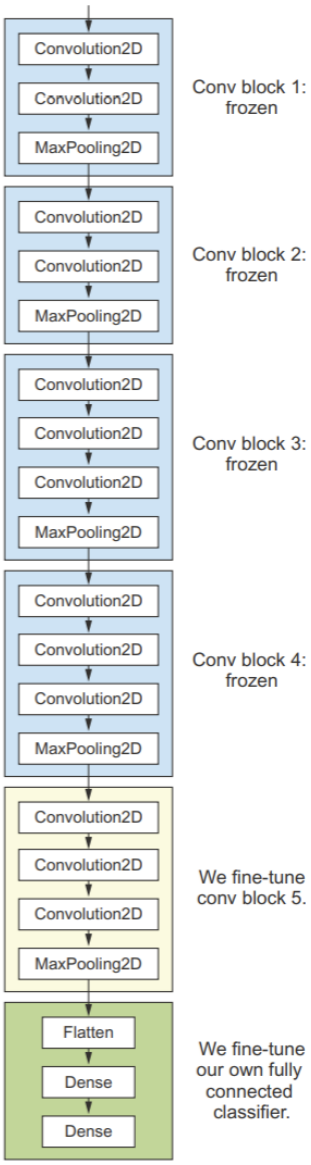


Figure 4.17: Fine tuning

The size and parameters of the model are shown in the figure below, and it can be seen that the trainable parameters are increased to 7,604,225.

```
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 vgg16 (Functional)          (None, 2, 2, 512)         14714688  
 flatten_12 (Flatten)        (None, 2048)              0         
 dense_25 (Dense)            (None, 256)               524544    
 dense_24 (Dense)            (None, 1)                 257       
=================================================================
Total params: 15239489 (58.13 MB)
Trainable params: 7604225 (29.01 MB)
Non-trainable params: 7635264 (29.13 MB)
_________________________________________________________________
```

Figure 4.18: Summary of Model 5.3

The training loss decreased gradually with the increase of the number of training rounds, indicating that the performance of the model on the training data gradually improved. The loss value gradually decreased from about 0.70 to close to 0.60. The validation loss decreased slightly at the beginning of the training, but then gradually increased and stabilized, the loss value gradually increased from about 0.70 to close to 0.75.

The training accuracy rate increased gradually with the increase of the number of training rounds, from about 0.60 to close to 0.70, indicating that the classification ability of the model on the training data was gradually enhanced. Validation accuracy increased slightly at the beginning of training, but then gradually decreased and stabilized, from about 0.63 to close to 0.61.

The training loss and accuracy of the model perform well in the process of training and validation, but the increase of validation loss and the decrease of accuracy indicate that the model has poor generalization ability on validation data, and there is overfitting problem.

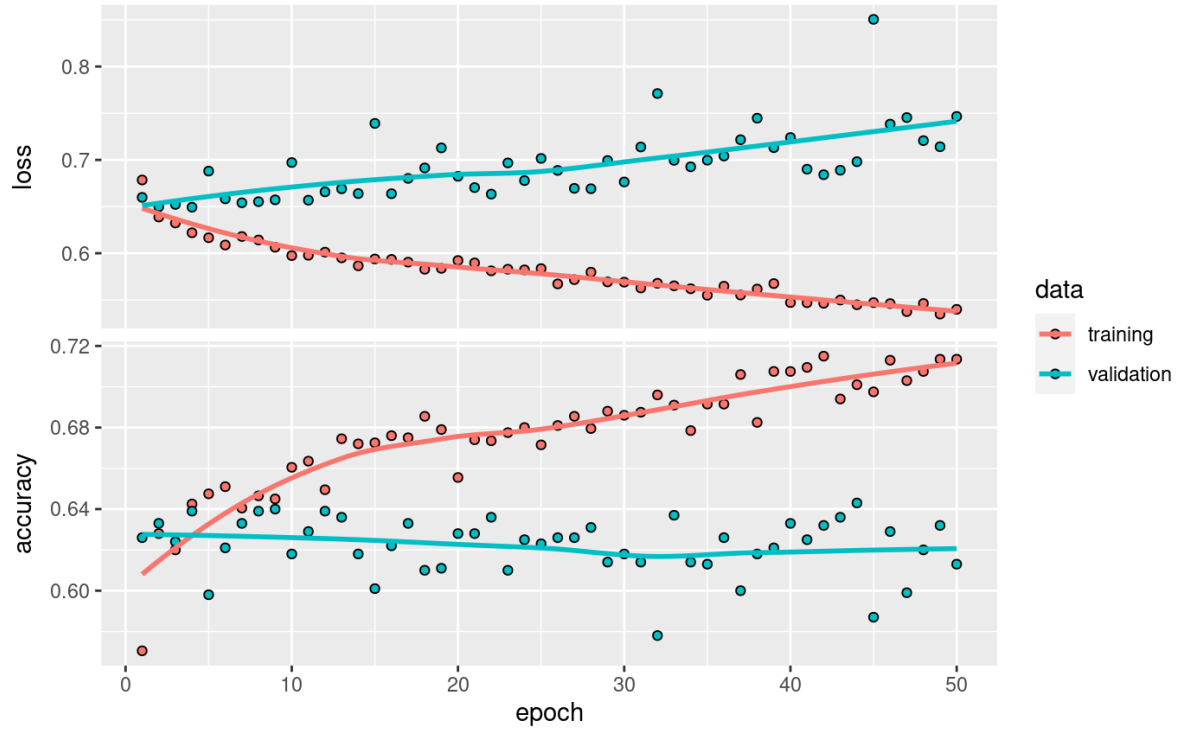


Figure 4.19: Training indicators and validation indicators of Model 5.3

4.6.3 Model 5.4 Run from end to end

Because the previous model is still not optimal, the last model will attempt to run from start to finish with 15,239,489 parameters

```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 2, 2, 512)         14714688
 flatten_13 (Flatten)        (None, 2048)              0
 dense_27 (Dense)            (None, 256)               524544
 dense_26 (Dense)            (None, 1)                 257
=================================================================
Total params: 15239489 (58.13 MB)
Trainable params: 15239489 (58.13 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 4.20: Summary of Model 5.4

The training loss decreased gradually with the increase of the number of training rounds, from about 0.70 to about 0.55, indicating that the model's performance on the training data gradually improved. The validation loss changed little at the beginning, and then gradually increased from about 0.65 to about 1.1, indicating that the performance of the model on the validation data gradually deteriorated.

The training accuracy rate increased gradually with the increase of the number of training rounds, from about 0.55 to about 0.70, indicating that the classification ability of the model on the training data was gradually enhanced. Validation accuracy rose slightly in the beginning, but then gradually declined and eventually stabilized, with accuracy rising from about 0.60 to close to 0.62.

There is still a certain degree of overfitting.

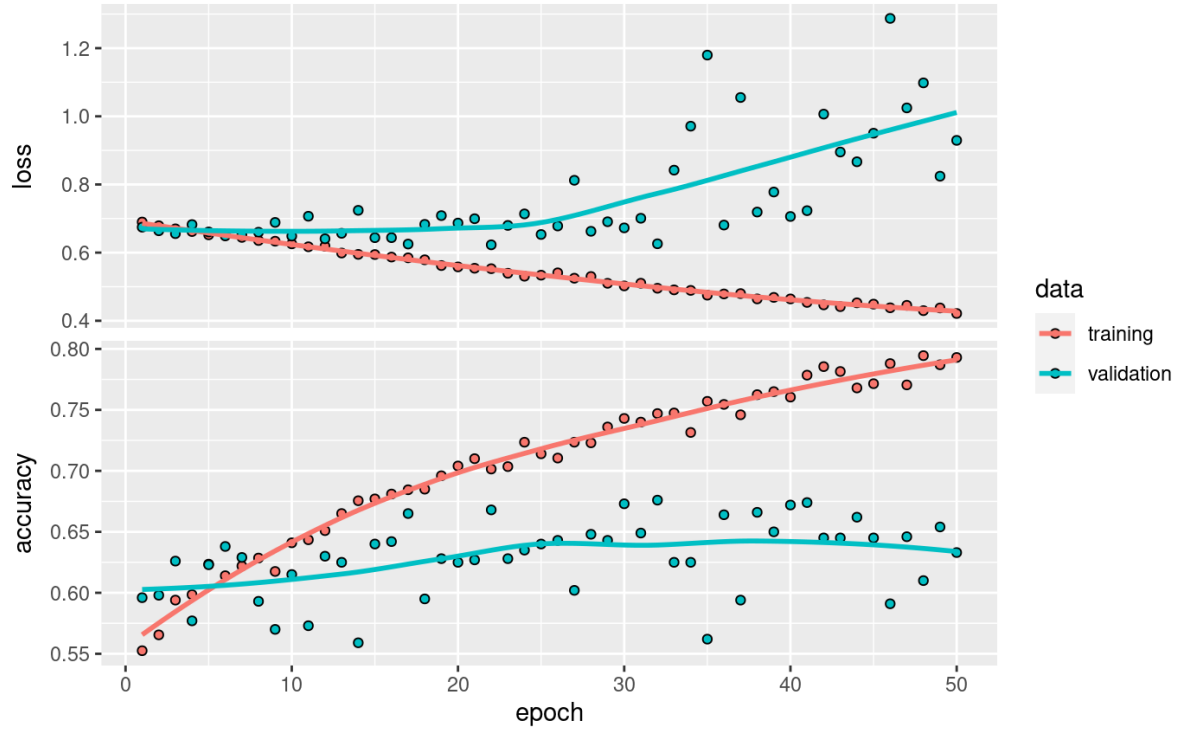


Figure 4.21: Training indicators and validation indicators of Model 5.4

Due to the limitations of the computer GPU, the image processing task is left to the UPC server. UPC servers have powerful computing and image processing capabilities to efficiently handle large data sets and complex tasks, increasing speed and accuracy while reducing the burden on local computers. This can optimize resource utilization and improve overall work efficiency.

4.7 Expand the size of medical images

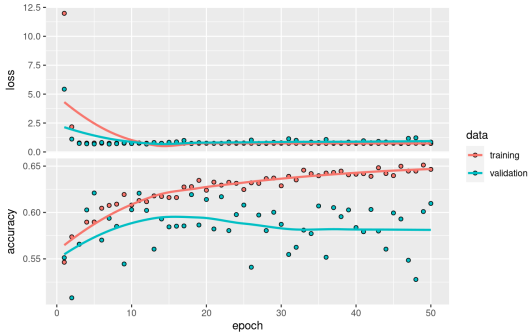
Since the model performs well on cats and dogs, after the problem of model setting is eliminated, in order to improve the accuracy, we will increase the amount of data and retrain in this section. Because the VGG16 models perform well, and the accuracy tends to improve, we will only train models about VGG16 after increasing the amount of data.

The specific size of the data increases to:

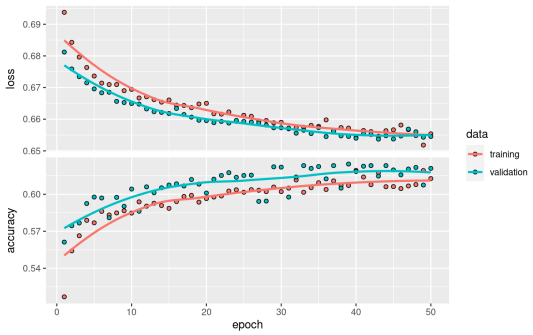
	Train	Validation	Test	Total
MSI	4,000	2,000	2,000	8,000
MSS	4,000	2,000	2,000	8,000
Total	8,000	4,000	4,000	16,000

Table 2: Data size 2

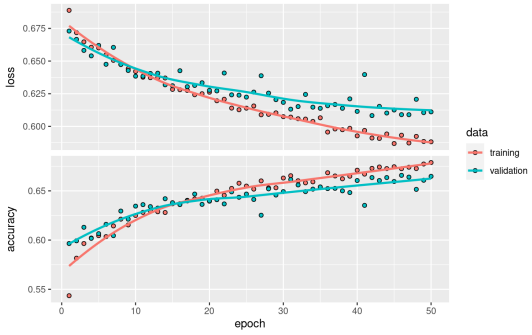
The resulting graphs shows that the accuracy rate has improved, almost reaching 0.7. The set test is reached to 0.675.



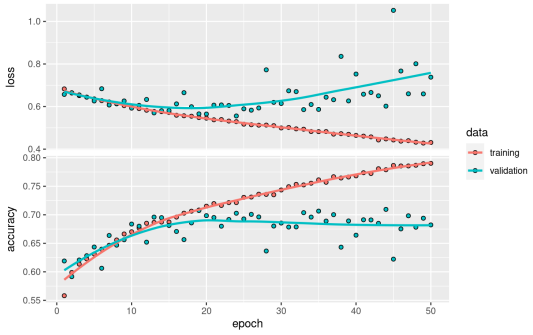
(a) Model 5.1



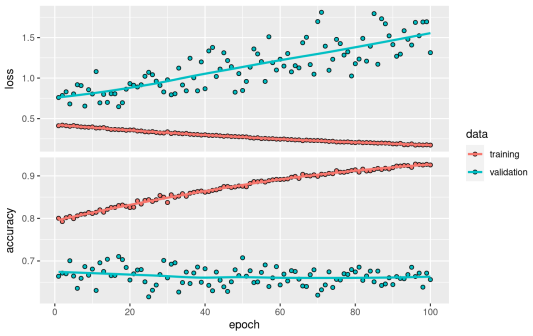
(b) Model 5.2



(c) Model 5.3



(d) Model 5.4 with 50 batches



(e) Model 5.4 with 100 batches

Figure 4.22: VGG16 models training on expanded size

```
> model5.4 %>% evaluate_generator(test_generator,steps = 50)
50/50 [==============================] - 2s 35ms/step - loss: 1.3483 - accuracy: 0.6750
    loss accuracy
1.348277 0.675000
```

Figure 4.23: Model5.4 test

4.8 Summary of all the models

This section will list the test accuracy of all models. It can be seen from the table 3 that Model 5 series generally better than the first four models, especially when the test sample size is increased. We can surmise that the accuracy rate will be even better when the sample size is increased.

Models	Test Accuracy
Size of test = 1000	
Model 1	0.525
Model 2	0.510
Model 3	0.530
Model 4	0.525
Model 5.1	0.590
Model 5.2	0.612
Model 5.3	0.625
Model 5.4	0.630
Size of test = 4000	
Model 5.1	0.575
Model 5.2	0.606
Model 5.3	0.656
Model 5.4	0.675

Table 3: Test accuracy of different Models

4.9 PCA, t-SNE, UMAP

After obtainning the accuracyin this section, will analyze the output of the flatten_3 layer of our neural network model 5.4 to assess its effectiveness in distinguishing between MSI and MSS samples. The goal is to determine the extent to which the model can separate these two categories in a lower-dimensional space. We employ three popular dimensionality reduction techniques: PCA (Principal Component Analysis), t-SNE (t-Distributed Stochastic Neighbor Embedding)[8], and UMAP (Uniform Manifold Approximation and Projection)[1]. These methods help us visualize the clustering behavior and separability of the data.

```
flatten_layer_name <- "flatten_3"
flatten_layer_model <- keras_model(inputs = model5.4.1$input, outputs = get_
    layer(model5.4.1, flatten_layer_name)$output)
```

```
all_flatten_output <- NULL
all_labels <- NULL

while(TRUE) {
  batch <- generator_next(train_generator)
  input_data <- batch[[1]]
  labels <- batch[[2]]

  if (is.null(all_flatten_output)) {
    all_flatten_output <- predict(flatten_layer_model, input_data)
    all_labels <- labels
  } else {
    all_flatten_output <- rbind(all_flatten_output, predict(flatten_layer_model,
        input_data))
    all_labels <- c(all_labels, labels)
  }
  if (length(all_labels) >= train_generator$samples) {
    break
  }
}
all_flatten_output <- all_flatten_output[, apply(all_flatten_output, 2, var) !=
   0]
```

Listing 1: Flatten_3

4.9.1 PCA

First, we will check with the PCA with a bar chart and two-dimensional scatter plot with labels.

This bar chart (Figure 4.24)shows the variance distribution in the results of principal component analysis. The X-axis represents the principal component and the Y-axis represents the variance of each principal component. As can be seen, the first few principal components explain most of the variance of the data, and the variance of the subsequent principal components gradually decreases.

```
pca_result <- prcomp(all_flatten_output)
plot(pca_result)
a <- summary(pca_result)
pca_data <- data.frame(pca_result$x)
pca_data$label <- as.factor(all_labels)

ggplot(pca_data, aes(x = PC1, y = PC2, color = label)) +
  geom_point() +
  scale_color_manual(values = c("0" = "red", "1" = "black")) +
  theme_minimal() +
  labs(title = "PCA of Flatten_3 Layer Output with Labels",
       x = "PC1",
       y = "PC2") +
  geom_hline(yintercept = 0, color = "blue") +
```

```
  geom_vline(xintercept = 0, color = "blue")
```

Listing 2: PCA

This graph shows that the high dimensional space of data can be explained by several major components. Typically, the first few principal components retain the main information of the data, while the subsequent principal components contain less information. This is very useful for reducing and feature selection, and can reduce the computational complexity by reducing the dimensions while maintaining the main features of the data.

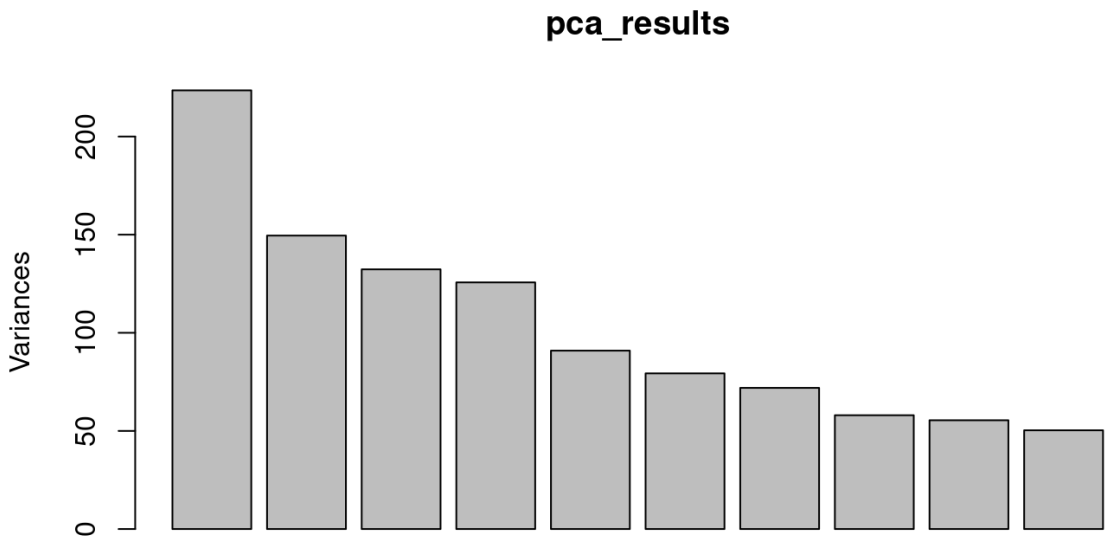


Figure 4.24: PCA

This two-dimensional scatter plot shows the results of PCA dimensionality reduction on the output of Flatten_3 layer. The X and Y axes represent PC1 and PC2, respectively. The red and black dots represent data points labeled 0 and 1, respectively.

As can be seen from this figure, after dimensionality reduction through PCA, the data points labeled 0 and 1 have a certain separation on the two-dimensional plane, but the separation effect is not very obvious. This shows that there is a certain overlap of samples of different categories in the output feature space of Flatten_3 layer. This may mean that the model needs to be further tweaked or more features extracted to better distinguish between different classes of samples.

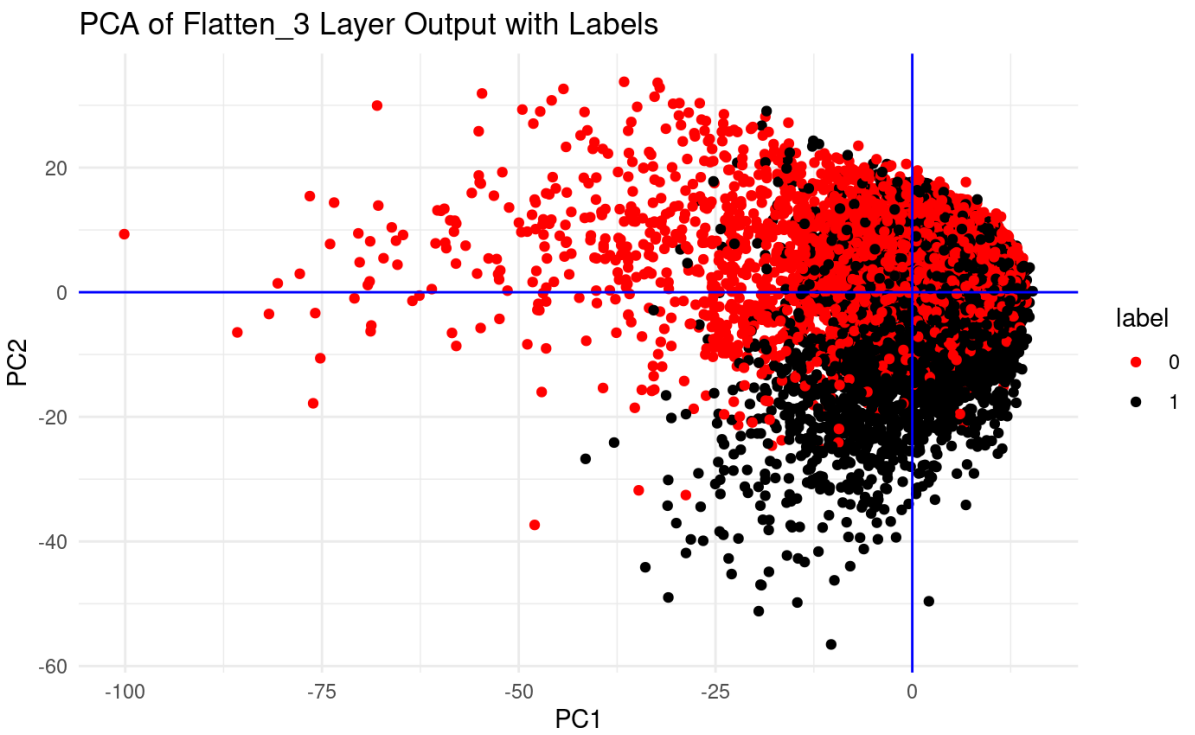


Figure 4.25: PCA with label

4.9.2 t-SNE

The same thing we do on t-SNE. This plot illustrates the results of applying t-SNE to the output of the flatten_3 layer from model 5.4. Each point in the plot represents a sample, with the X-axis and Y-axis representing the two t-SNE dimensions. With labels, red for MS and black for MSI.

X-axis and Y-axis represent the first and second dimensions obtained from the t-SNE algorithm. t-SNE attempts to place similar points near each other in this 2D space, preserving local structure from the high-dimensional data.

The t-SNE plot shows that there is some degree of separation between MSI and MSS samples, but there is also considerable overlap. This indicates that while t-SNE has been able to capture some of the differences between the two classes, these differences are not sufficiently distinct in the two-dimensional space to achieve clear separation.

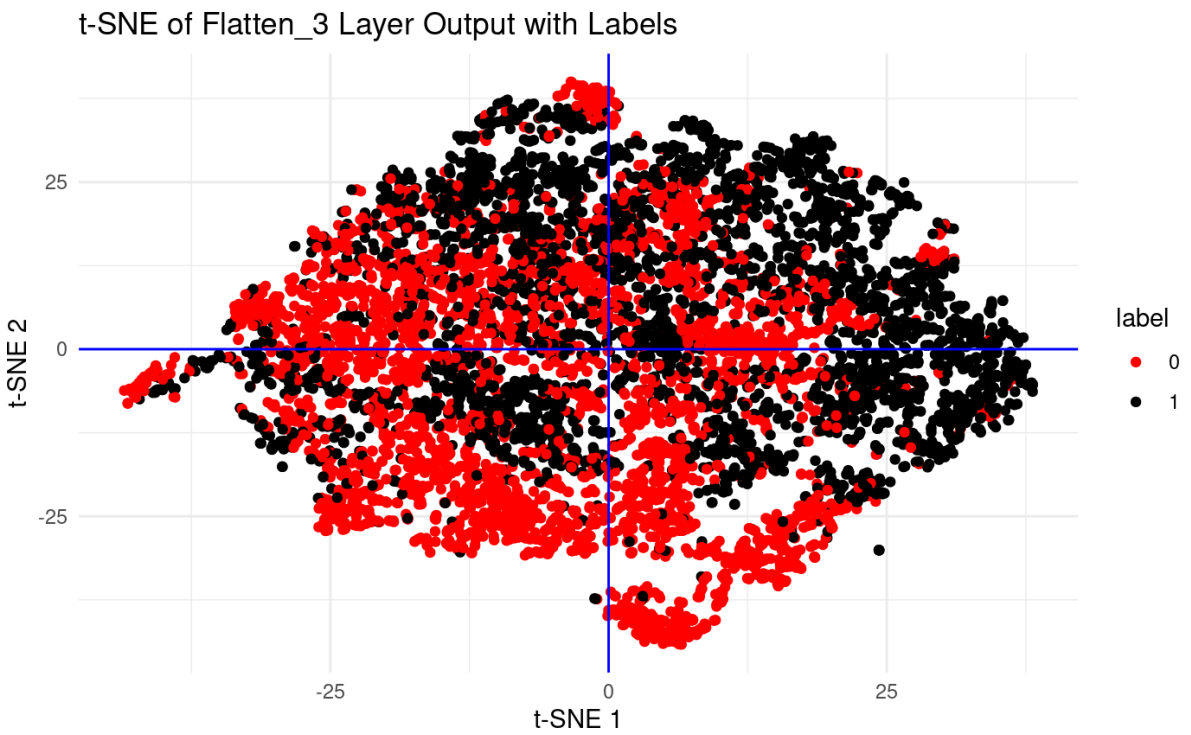


Figure 4.26: t-SNE

4.9.3 UMAP

This plot illustrates the results of applying UMAP to the output of the flatten_3 layer from model 5.4.

The UMAP plot shows a moderate degree of separation between MSI and MSS samples, but there is still considerable overlap. This indicates that UMAP has been also able to capture some of the differences between the two classes, but still need more further reaserch.

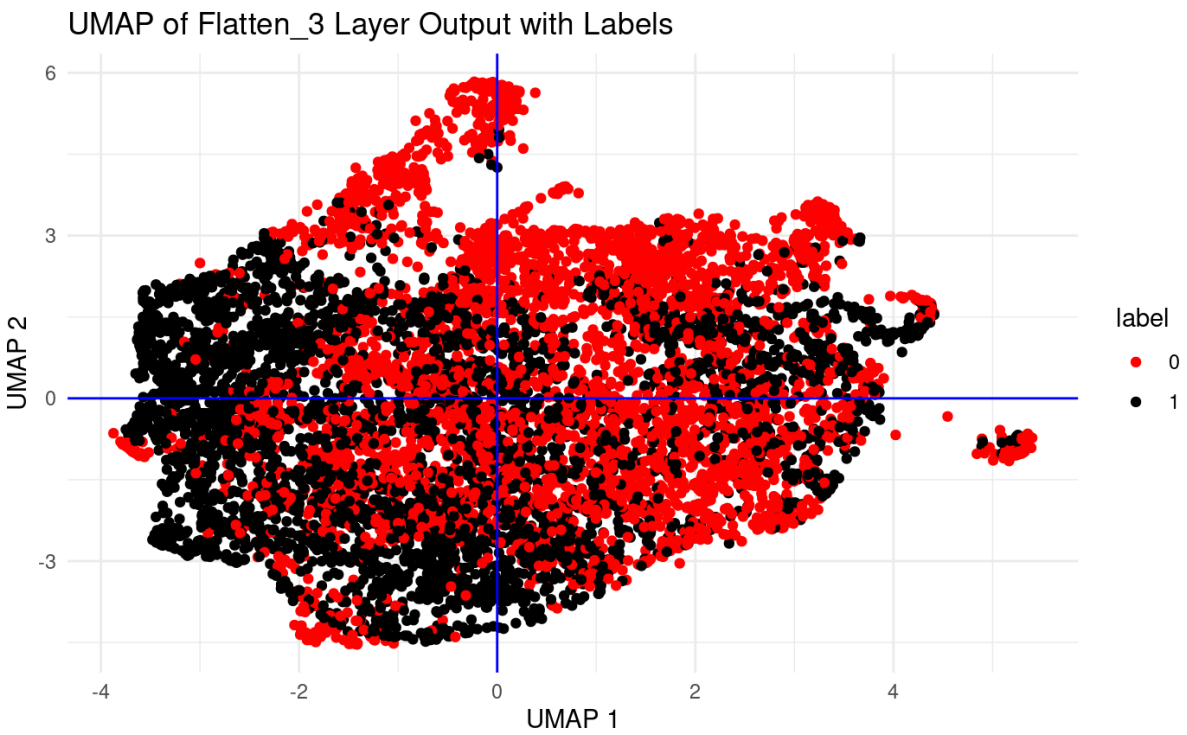


Figure 4.27: UMAP

5 Discussion

This study has achieved a lot in using convolutional neural networks to classify microsatellite instability and stability in histological images of gastrointestinal cancer, but there are still some limitations that need to be improved in future studies:

1. Dataset size: Although this study used a relatively large dataset, the sample size is still insufficient, which may affect the generalization ability of the model. Future studies may consider collecting more samples to improve the robustness and accuracy of the model, as well as to study whether the smallest sample size can reach the specified precise value.

2. Image preprocessing: Although we have standardized and expanded image processing, there may be differences in the preprocessing methods of images from different sources, which may lead to unstable performance of the model in practical applications. Further research can explore more consistent and optimized pretreatment methods.

3. Model complexity: Although deep learning models have strong learning ability, their high complexity also brings high computing costs and long training time. Future research could explore more efficient model architectures or optimization algorithms to reduce the consumption of computing resources.

4. Feature interpretability: Deep learning models are often considered ”black box” models that lack transparency into the decision-making process. Although the CNN model in this study can effectively classify MSI and MSS, its feature extraction and decision mechanism are not clear. Future studies could incorporate interpretability techniques to improve model transparency and interpretability.

6 Conclusion

This study explores the application of CNN in medical image classification, especially the effectiveness of MSI and MSS in histological images of gastrointestinal cancer. The results show that the deep CNN architecture can effectively predict MSI and MSS a, providing clinicians with a reliable tool to identify the microsatellite stability of tumor tissue.

First of all, it has been proved by experiments that CNN has relatively high accuracy in classifying MSI and MSS of tumor tissue about 70% of accuracy. The pre-trained VGG16 model combined with feature extraction and fine-tuning techniques significantly improved the classification effect from 50% to 70%. Data enhancement and Dropout regularization techniques have also shown an important role in reducing overfitting.

Second, pre-trained models also perform quite well on small data sets, and through transfer learning, these models can quickly adapt to new classification tasks and improve accuracy. However, the high complexity of deep learning models also brings high computational costs and long training time, and future research can explore more efficient model architectures or optimization algorithms.

Finally, the results of this study perform well in the experimental environment, but there are still challenges to be faced in the actual clinical application, such as the difference in image quality between different hospitals and equipment, and individual patient differences. Therefore, future studies should focus on the performance of the model under multi-center and multi-device conditions, and conduct clinical validation.

Overall, this study demonstrates the potential of convolutional neural networks in medical image classification, particularly in microsatellite instability and stability classification tasks for tumor tissue. Through reasonable model design and optimization, the classification accuracy can be significantly improved, and an effective tool for medical image analysis can be provided.

7 References

[1] Etienne Becht, Leland McInnes, John Healy, et al. “Dimensionality reduction for visualizing single-cell data using UMAP”. In: *Nature Biotechnology* 37 (2019), pp. 38–44. DOI: 10.1038/nbt.4314.

[2] Francois Chollet and J. J. Allaire. *Deep Learning with R*. 1st. USA: Manning Publications Co., 2018. ISBN: 161729554X.

[3] Manjunath Jogin et al. “Feature Extraction using Convolution Neural Networks (CNN) and Deep Learning”. In: *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*. 2018, pp. 2319–2323. DOI: 10.1109/RTEICT42901.2018.9012507.

[4] Jakob N Kather, Adam T Pearson, Niels Halama, et al. “Deep learning can predict microsatellite instability directly from histology in gastrointestinal cancer”. In: *Nature Medicine* 25.7 (2019), pp. 1054–1056. DOI: 10.1038/s41591-019-0462-y. URL: https://doi.org/10.1038/s41591-019-0462-y.

[5] Jakob Nikolas Kather. *Histological images for MSI vs. MSS classification in gastrointestinal cancer, FFPE samples*. Version 1, Feb 7, 2019. 2019. URL: https://doi.org/10.5281/zenodo.2530835.

[6] KA Tran, O Kondrashova, A Bradley, et al. “Deep learning in cancer diagnosis, prognosis and treatment selection”. In: *Genome Medicine* 13 (2021), p. 152. DOI: 10.1186/s13073-021-00968-x. URL: https://doi.org/10.1186/s13073-021-00968-x.

[7] M Unger and JN Kather. “Deep learning in cancer genomics and histopathology”. In: *Genome Medicine* 16.44 (2024). DOI: 10.1186/s13073-024-01315-6. URL: https://doi.org/10.1186/s13073-024-01315-6.

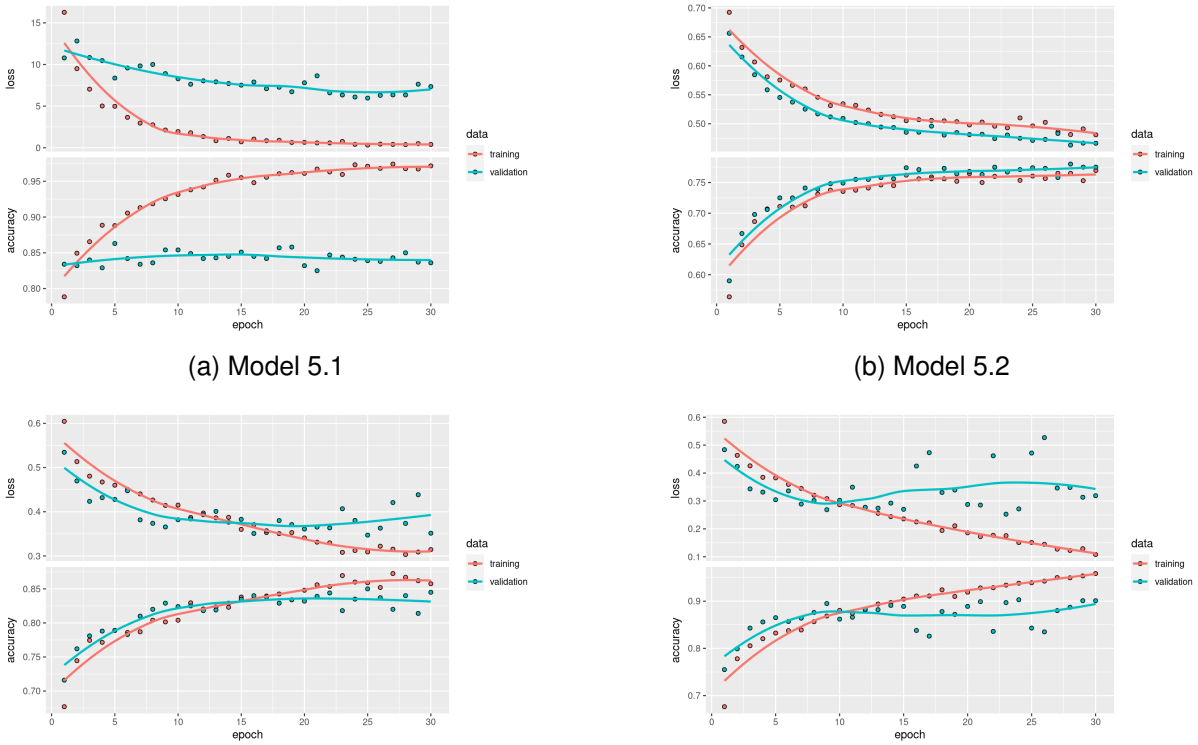
[8] Laurens Van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE.” In: *Journal of machine learning research* 9.11 (2008).

8 Appendix

8.1 Training on the cats and dogs images

In order to verify whether there are problems with our model, we will train our written VGG16 in the most classic cats and dogs[2] images to determine whether adjustment is needed.

As can be seen from the figure, there are no problems with the model. All of the models achieved at least 0.7 accuracy.



(a) Model 5.1

(b) Model 5.2

(c) Model 5.3

(d) Mode 5.4

Figure 8.1: VGG16 models training on the cats and dogs images

8.2 Codes

```
---
title: "borrador de TFG"
author: "Shengnan"
date: "2024-04-29"
output:
  html_document: default
  pdf_document:
    latex_engine: xelatex
    keep_tex: true
---

```{r setup, include=FALSE}
```
```

````
knitr::opts_chunk$set(echo = TRUE)
```

# datos

```{r}
library(keras)
library(tensorflow)
library(tfdatasets)
```

rename and split into train, test, validation carpet

```{r}

base_dir <- "~/TFG/msi_mss"
#dir.create(base_dir)

train_dir<-file.path(base_dir,"train",fsep ="/")
#dir.create(train_dir)

validation_dir<-file.path(base_dir,"validation",fsep ="/")
#dir.create(validation_dir)

test_dir<-file.path(base_dir,"test",fsep ="/")
#dir.create(test_dir)

train_MSI_dir<-file.path(train_dir,"MSI",fsep ="/")
#dir.create(train_MSI_dir)
train_MSS_dir<-file.path(train_dir,"MSS",fsep ="/")
#dir.create(train_MSS_dir)

validation_MSI_dir<-file.path(validation_dir,"MSI",fsep ="/")
#dir.create(validation_MSI_dir)
validation_MSS_dir<-file.path(validation_dir,"MSS",fsep ="/")
#dir.create(validation_MSS_dir)

test_MSI_dir<-file.path(test_dir,"MSI",fsep ="/")
#dir.create(test_MSI_dir)
test_MSS_dir<-file.path(test_dir,"MSS",fsep ="/")
#dir.create(test_MSS_dir)

# images_dir <- "~/TFG/base/MSI"
# files <- list.files(images_dir, pattern = "\\png$", full.names = FALSE)
# num_files <- length(files)
# new_names <- sprintf("MSI.%d.jpg", 1:num_files)
# print(new_names)
# #
# if (num_files > 0) {
#     new_names <- sprintf("MSI.%d.jpg", 1:num_files)
#     new_files <- file.path(images_dir, new_names)
````

```
#     result <- mapply(file.rename, from = file.path(images_dir, files), to =
    new_files)
#     print(result)
# } else {
#     print("No JPG files found in the specified directory.")
# }
# #
# images2_dir <- "~/TFG/base/MSS"
# files2 <- list.files(images2_dir, pattern = "\\png$", full.names = FALSE)
# num_files2 <- length(files2)
# new_names2 <- sprintf("MSS.%d.jpg", 1:num_files2)
# print(new_names2)
#
# if (num_files2 > 0) {
#     new_names2 <- sprintf("MSS.%d.jpg", 1:num_files2)
#     new_files2 <- file.path(images2_dir, new_names2)
#     result <- mapply(file.rename, from = file.path(images2_dir, files2), to =
    new_files2)
#     print(result)
# } else {
#     print("No JPG files found in the specified directory.")
# }
#
#
original_dataset_dir_MSI <-  "~/TFG/base/MSI"
original_dataset_dir_MSS <-  "~/TFG/base/MSS"

########### MSI 1-2000 ###########
# fnames <- paste0("MSI.",1:1000,".jpg")
# file.copy(file.path(original_dataset_dir_MSI,fnames),
#           file.path(train_MSI_dir))
#
# fnames <- paste0("MSI.",1001:1500,".jpg")
# file.copy(file.path(original_dataset_dir_MSI,fnames),
#           file.path(validation_MSI_dir))
#
# fnames <- paste0("MSI.",1501:2000,".jpg")
# file.copy(file.path(original_dataset_dir_MSI,fnames),
#           file.path(test_MSI_dir))

########### MSS 1-2000 ###########
# fnames <- paste0("MSS.",1:1000,".jpg")
# file.copy(file.path(original_dataset_dir_MSS,fnames),
#           file.path(train_MSS_dir))
#
# fnames <- paste0("MSS.",1001:1500,".jpg")
# file.copy(file.path(original_dataset_dir_MSS,fnames),
#           file.path(validation_MSS_dir))
#
# fnames <- paste0("MSS.",1501:2000,".jpg")
# file.copy(file.path(original_dataset_dir_MSS,fnames),
```

```
#           file.path(test_MSS_dir))

########### MSI 5001-12500  ###########
# fnames <- paste0("MSI.",5000:7999,".jpg")
# file.copy(file.path(original_dataset_dir_MSI,fnames),
#           file.path(train_MSI_dir))
#
# fnames <- paste0("MSI.",9001:10500,".jpg")
# file.copy(file.path(original_dataset_dir_MSI,fnames),
#           file.path(validation_MSI_dir))
#
# fnames <- paste0("MSI.",11001:12500,".jpg")
# file.copy(file.path(original_dataset_dir_MSI,fnames),
#           file.path(test_MSI_dir))

########### MSS 5001-12500  ###########
# fnames <- paste0("MSS.",5000:7999,".jpg")
# file.copy(file.path(original_dataset_dir_MSS,fnames),
#           file.path(train_MSS_dir))
#
# fnames <- paste0("MSS.",9001:10500,".jpg")
# file.copy(file.path(original_dataset_dir_MSS,fnames),
#           file.path(validation_MSS_dir))
#
# fnames <- paste0("MSS.",11001:12500,".jpg")
# file.copy(file.path(original_dataset_dir_MSS,fnames),
#           file.path(test_MSS_dir))

```
```


CNN

Modelo 1

```{r}

model1 <- keras_model_sequential() %>%
  layer_conv_2d(filter = 32, kernel_size = c(7,7) , activation = "relu",
                input_shape = c(224,224,3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filter = 64 , kernel_size = c(7,7) , activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filter = 128 , kernel_size = c(5,5) , activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filter = 128 , kernel_size = c(3,3) , activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_flatten() %>%
  layer_dense(units = 512 , activation = "relu") %>%
  layer_dense(units = 1 , activation = "sigmoid")
```
```

```
summary(model1)



# input_layer<-layer_input(c(150, 150, 3))
# x=layer_conv_2d(filters = 32, kernel_size = c(7, 7))(input_layer)
# x=layer_activation(activation = "relu")(x)
# x=layer_max_pooling_2d(pool_size =c(2,2))(x)
# x=layer_conv_2d(filters = 64, kernel_size = c(5, 5))(x)
# x=layer_activation(activation = "relu")(x)
# x=layer_max_pooling_2d(pool_size =c(2,2))(x)
# x=layer_conv_2d(filters = 64, kernel_size = c(3, 3))(x)
# x=layer_activation(activation = "relu")(x)
# x=layer_max_pooling_2d(pool_size =c(2,2))(x)
# x=layer_flatten(x)
# x=layer_dense(units = 128)(x)
# x=layer_activation(activation = "relu")(x)
# x=layer_dense(units = 1)(x)
# output_layer=layer_activation(activation = "sigmoid")(x)
```



```{r}

train_datagen <- image_data_generator(rescale = 1/255)
validation_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory (train_dir,
                                               train_datagen,
                                               target_size = c(224, 224),
                                               batch_size = 20,
                                               class_mode = "binary"
                                               )
validation_generator <- flow_images_from_directory (validation_dir,
                                                    validation_datagen,
                                                    target_size = c(224, 224),
                                                    batch_size = 20,
                                                    class_mode = "binary"
                                                    )
batch <- generator_next(train_generator)
str(batch)

# train_datagen <- image_data_generator(rescale = 1/255)
# validation_datagen <- image_data_generator(rescale = 1/255)
#
# train_generator <- flow_images_from_directory (train_dir,
#                                                train_datagen,
#                                                target_size = c(224, 224),
#                                                batch_size = 80,#20
```

```
#                                                    class_mode = "binary"
#                                                    )
# validation_generator <- flow_images_from_directory (validation_dir,
#                                                     validation_datagen,
#                                                     target_size = c(224, 224),
#                                                     batch_size = 80,#20
#                                                     class_mode = "binary"
#                                                     )
# batch <- generator_next(train_generator)
# str(batch)


```

```{r}
model1 %>%  compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-4),
  metrics = c("accuracy")
)

history <- model1 %>% fit(
  train_generator,
  steps_per_epoch = 100,
  epochs = 50,
  validation_data = validation_generator,
  validation_steps = 50,
  run_eagerly = TRUE
)

model1 %>% save_model_hdf5("MSI_MSS_1.h5")
model1 <- load_model_hdf5("MSI_MSS_1.h5")
```
```{r}
plot(history)
```
overfitting

```
# Modelo 2 64*64
```{r}
model2.1<- keras_model_sequential() %>%
  layer_conv_2d(filter = 32, kernel_size = c(7,7) , activation = "relu",
                input_shape = c(64,64,3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filter = 64 , kernel_size = c(5,5) , activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filter = 128 , kernel_size = c(5,5) , activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filter = 128 , kernel_size = c(3,3) , activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_flatten() %>%
```
```

```
  layer_dense(units = 512 , activation = "relu") %>%
  layer_dense(units = 1 , activation = "sigmoid")


model2 <- keras_model_sequential() %>%
  layer_conv_2d(filter = 32, kernel_size = c(7,7) , activation = "relu",
                input_shape = c(64,64,3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filter = 64 , kernel_size = c(5,5) , activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filter = 128 , kernel_size = c(5,5) , activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_flatten() %>%
  layer_dense(units = 512 , activation = "relu") %>%
  layer_dense(units = 1 , activation = "sigmoid")
summary(model2.1)
summary(model2)
```
```
```{r}
train_datagen <- image_data_generator(rescale = 1/255)
validation_datagen <- image_data_generator(rescale = 1/255)
train_generator <- flow_images_from_directory (train_dir,
                                               train_datagen,
                                               target_size = c(64, 64),
                                               batch_size = 20,
                                               class_mode = "binary"
                                               )
validation_generator <- flow_images_from_directory (validation_dir,
                                                    validation_datagen,
                                                    target_size = c(64, 64),
                                                    batch_size = 20,
                                                    class_mode = "binary"
                                                    )
batch <- generator_next(train_generator)
str(batch)

# train_generator <- flow_images_from_directory (train_dir,
#                                                train_datagen,
#                                                target_size = c(64, 64),
#                                                batch_size = 80,
#                                                class_mode = "binary"
#                                                )
# validation_generator <- flow_images_from_directory (validation_dir,
#                                                     validation_datagen,
#                                                     target_size = c(64, 64),
#                                                     batch_size = 80,
#                                                     class_mode = "binary"
#                                                     )
# batch <- generator_next(train_generator)
# str(batch)
```
```

```{r}
model2 %>%  compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-4),
  metrics = c("accuracy")
)
history2 <- model2 %>% fit(
  train_generator,
  steps_per_epoch = 100,
  epochs = 50,
  validation_data = validation_generator,
  validation_steps = 50,
  run_eagerly = TRUE
)

# history2 <- model2 %>% fit(
#   train_generator,
#   steps_per_epoch = 60,
#   epochs = 20,
#   validation_data = validation_generator,
#   validation_steps = 50,
#   run_eagerly = TRUE
# )

model2 %>% save_model_hdf5("MSI_MSS_2.h5")
model2 <- load_model_hdf5("MSI_MSS_2.h5")

plot(history2)
```

Modelo 3 DROP OUT AND AUGMENTATION

```{r}

model3 <- keras_model_sequential() %>%
  layer_conv_2d(filter = 32, kernel_size = c(7,7) , activation = "relu",
                input_shape = c(64,64,3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filter = 64 , kernel_size = c(5,5) , activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filter = 128 , kernel_size = c(5,5) , activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_flatten() %>%
  layer_dropout(rate=0.5) %>%
  layer_dense(units = 512 , activation = "relu") %>%
  layer_dense(units = 1 , activation = "sigmoid")
summary(model3)



# input_layer<-layer_input(c(150, 150, 3))
```

```
# x=layer_conv_2d(filters = 32, kernel_size = c(7, 7))(input_layer)
# x=layer_activation(activation = "relu")(x)
# x=layer_max_pooling_2d(pool_size =c(2,2))(x)
# x=layer_conv_2d(filters = 64, kernel_size = c(5, 5))(x)
# x=layer_activation(activation = "relu")(x)
# x=layer_max_pooling_2d(pool_size =c(2,2))(x)
# x=layer_conv_2d(filters = 64, kernel_size = c(3, 3))(x)
# x=layer_activation(activation = "relu")(x)
# x=layer_max_pooling_2d(pool_size =c(2,2))(x)
# x=layer_flatten(x)
# x=layer_dense(units = 128)(x)
# x=layer_activation(activation = "relu")(x)
# x=layer_dense(units = 1)(x)
# output_layer=layer_activation(activation = "sigmoid")(x)
```

```{r}
datagen <- image_data_generator(
  rescale = 1/255,
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = TRUE,
)
```

```{r}
test_datagen <- image_data_generator(rescale = 1/255)
train_generator <- flow_images_from_directory (train_dir,
                                               datagen,
                                               target_size = c(64, 64),
                                               batch_size = 20,
                                               class_mode = "binary"
                                               )
validation_generator <- flow_images_from_directory (validation_dir,
                                                    test_datagen,
                                                    target_size = c(64, 64),
                                                    batch_size = 20,
                                                    class_mode = "binary")

# train_generator <- flow_images_from_directory (train_dir,
#                                                datagen,
#                                                target_size = c(64, 64),
#                                                batch_size = 80,
#                                                class_mode = "binary"
#                                                )
# validation_generator <- flow_images_from_directory (validation_dir,
#                                                     test_datagen,
#                                                     target_size = c(64, 64),
```

```
#                                                    batch_size = 80,
#                                                    class_mode = "binary")


```

```{r}
model3 %>%  compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-4),
  metrics = c("accuracy")
)

history3 <- model3 %>% fit(
  train_generator,
  steps_per_epoch = 100,
  epochs = 50,
  validation_data = validation_generator,
  validation_steps = 50,
)

model3 %>% save_model_hdf5("MSI_MSS_3.h5")
model3 <- load_model_hdf5("MSI_MSS_3.h5")
```
```{r}
plot(history3)
```

Modelo 4 NORMALIZATION scale(x)

```{r}
model4 <- keras_model_sequential() %>%
  layer_conv_2d(filter = 32, kernel_size = c(7,7) , activation = "relu",
                input_shape = c(64,64,3)) %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filter = 64 , kernel_size = c(5,5) , activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filter = 128 , kernel_size = c(5,5) , activation = "relu") %>%
  layer_batch_normalization() %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_flatten() %>%
  layer_dropout(rate=0.5) %>%
  layer_dense(units = 512 , activation = "relu") %>%
  layer_dense(units = 1 , activation = "sigmoid")

summary(model4)
```

```{r}
model4 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-4),
  metrics = c("accuracy")
)
```


```{r}
history4 <- model4 %>% fit (
  train_generator,
  steps_per_epoch = 100,
  epochs = 50,  # 20
  validation_data = validation_generator,
  validation_steps = 10)
```


```{r}
plot(history4)
```

VGG16

Plases see below the VGG16 model.
```{r}
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(64,64,3)
)
conv_base
```
Model 5.1 and Model 5.2 Feature extraction
```{r}
# datagen <- image_data_generator(rescale = 1/255)
# batch_size <- 20
#
# extract_features <- function(directory,sample_count){
#   features <- array(0,dim = c(sample_count,4,4,512))
#   labels <- array(0,dim = c(sample_count))
#
#   generator <- flow_images_from_directory(
#     directory = directory,
#     generator = datagen,
#     target_size = c(64,64),
#     batch_size = batch_size,
#     class_mode = "binary"
#   )
#   i <- 0
```

```
#   while(TURE){
#     batch <- generator_next(generator)
#     inputs_batch <- batch[[1]]
#     labels_batch <- batch[[2]]
#     features_batch <- conv_base %>% oredict(inputs_batch)
#
#     index_range <- ((i * batch_size)+1):((i+1)*batch_size)
#     features[index_range,,,] <- features_batch
#     labels[index_range] <- labels_batch
#
#     i <- i+1
#     if(i*batch_size >=sample_count)
#       break
#   }
#   list(
#     features=features,
#     labels = labels
#   )
# }
# train <- extract_features(train_dir,2000)
# validation <- extract_features(validation_dir,1000)
# test <- extract_features(test_dir,1000)
train_dataset <-image_dataset_from_directory(train_dir,
                                             image_size = c(64, 64),
                                             batch_size = 32)
validation_dataset <- image_dataset_from_directory(validation_dir,
                                                   image_size = c(64, 64),
                                                   batch_size = 32)
test_dataset <- image_dataset_from_directory(test_dir,
                                             image_size = c(64, 64),
                                             batch_size = 32)

get_features_and_labels <- function(dataset) {
  n_batches <- length(dataset)
  all_features <- vector("list", n_batches)
  all_labels <- vector("list", n_batches)
  iterator <- as_array_iterator(dataset)
  for (i in 1:n_batches) {
    c(images, labels) %<-% iter_next(iterator)
    preprocessed_images <- imagenet_preprocess_input(images)
    features <- conv_base %>% predict(preprocessed_images)
    all_labels[[i]] <- labels
    all_features[[i]] <- features
    }
  all_features <- listarrays::bind_on_rows(all_features)
  all_labels <- listarrays::bind_on_rows(all_labels)
  list(all_features, all_labels)
  }
c(train_features, train_labels) %<-% get_features_and_labels(train_dataset)
c(val_features, val_labels) %<-% get_features_and_labels(validation_dataset)
c(test_features, test_labels) %<-% get_features_and_labels(test_dataset)
```

```
dim(train_features)

inputs <- layer_input(shape = c(2, 2, 512))
outputs <- inputs %>%
  layer_flatten() %>%
  layer_dense(256) %>%
  layer_dropout(.5) %>%
  layer_dense(1, activation = "sigmoid")
model5.1 <- keras_model(inputs, outputs)
model5.1 %>% compile(loss = "binary_crossentropy",
                     optimizer = "rmsprop",
                     metrics = "accuracy")
summary(model5.1)
callbacks <- list(
  callback_model_checkpoint(
    filepath = "feature_extraction.keras",
    save_best_only = TRUE,
    monitor = "val_loss"
    )
  )
history5.1 <- model5.1 %>% fit(
  train_features, train_labels,
  epochs = 50,
  validation_data = list(val_features, val_labels),
  callbacks = callbacks
  )

plot(history5.1)
model5.1 %>% evaluate_generator(test_generator,steps = 80)
```

```{r}
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(64,64,3)
)

freeze_weights(conv_base)

unfreeze_weights(conv_base)
cat("This is the number of trainable weights before freezing,",
    "the conv base:", length(conv_base$trainable_weights),"\n")
freeze_weights(conv_base)
cat("This is the number of trainable weights after freezing,",
    "the conv base:", length(conv_base$trainable_weights),"\n")

# data_augmentation <- keras_model_sequential() %>%
#   layer_random_flip("horizontal") %>%
```

```
#   layer_random_rotation(0.1) %>%
#   layer_random_zoom(0.2)
# inputs <- layer_input(shape = c(64, 64, 3))
# outputs <- inputs %>%
#   data_augmentation() %>%
#   imagenet_preprocess_input() %>%
#   conv_base() %>%
#   layer_flatten() %>%
#   layer_dense(256) %>%
#   layer_dropout(0.5) %>%
#   layer_dense(1, activation = "sigmoid")

model5.2 <- keras_model_sequential() %>%
  conv_base %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
model5.2
model5.2 %>% evaluate_generator(test_generator,steps = 80)
```



```{r}
train_datagen <- image_data_generator(
  rescale = 1/255,
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = TRUE,
  fill_mode = "nearest"
)

test_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory (train_dir,
                                               train_datagen,
                                               target_size = c(64, 64),
                                               batch_size = 20,#80
                                               class_mode = "binary"
                                               )
validation_generator <- flow_images_from_directory (validation_dir,
                                                    test_datagen,
                                                    target_size = c(64, 64),
                                                    batch_size = 20,#80
                                                    class_mode = "binary")

model5.2 %>% compile(
  loss = "binary_crossentropy",
```

```
  optimizer = optimizer_rmsprop(learning_rate = 2e-5),
  metrics = c("accuracy")
)

history5.2 <- model5.2 %>% fit (
  train_generator,
  steps_per_epoch = 100,
  epochs = 50,  # 20
  validation_data = validation_generator,
  validation_steps = 50)

plot(history5.2)
```

Model 5.3 Fine tuning

```{r}
conv_base
freeze_weights(conv_base)
unfreeze_weights(conv_base,from = -4)

model5.3 <- keras_model_sequential() %>%
  conv_base %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
summary(model5.3)


model5.3 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-5),
  metrics = c("accuracy")
)

history5.3 <- model5.3 %>% fit (
  train_generator,
  steps_per_epoch = 100,
  epochs = 50,  # 20
  validation_data = validation_generator,
  validation_steps = 50)

plot(history5.3)
model5.3 %>% evaluate_generator(test_generator,steps = 80)
```

Model 5.4 Run from end to end

```{r}
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
```

```
  input_shape = c(64,64,3)
)
conv_base


model5.4 <- keras_model_sequential() %>%
  conv_base %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
summary(model5.4)


model5.4 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-5),
  metrics = c("accuracy")
)

history5.4 <- model5.4 %>% fit (
  train_generator,
  steps_per_epoch = 100,
  epochs = 50 ,  # 20
  validation_data = validation_generator,
  validation_steps = 50)

plot(history5.4)


test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,
  target_size = c(64,64),
  batch_size = 20,
  class_mode = "binary"
)

model5.4 %>% evaluate_generator(test_generator,steps = 50)
```


```
# Add more images
```{r}

base_dir <- "~/TFG/msi_mss"
#dir.create(base_dir)

train_dir<-file.path(base_dir,"train",fsep ="/")
#dir.create(train_dir)

validation_dir<-file.path(base_dir,"validation",fsep ="/")
```
```

```
#dir.create(validation_dir)

test_dir<-file.path(base_dir,"test",fsep ="/")
#dir.create(test_dir)

train_MSI_dir<-file.path(train_dir,"MSI",fsep ="/")
#dir.create(train_MSI_dir)
train_MSS_dir<-file.path(train_dir,"MSS",fsep ="/")
#dir.create(train_MSS_dir)

validation_MSI_dir<-file.path(validation_dir,"MSI",fsep ="/")
#dir.create(validation_MSI_dir)
validation_MSS_dir<-file.path(validation_dir,"MSS",fsep ="/")
#dir.create(validation_MSS_dir)

test_MSI_dir<-file.path(test_dir,"MSI",fsep ="/")
#dir.create(test_MSI_dir)
test_MSS_dir<-file.path(test_dir,"MSS",fsep ="/")
#dir.create(test_MSS_dir)

# original_dataset_dir_MSI <-  "~/TFG/base/MSI"
# original_dataset_dir_MSS <-  "~/TFG/base/MSS"
#
# ############ MSI 5001-12500  ############
# fnames <- paste0("MSI.",5001:8000,".jpg")
# file.copy(file.path(original_dataset_dir_MSI,fnames),
#           file.path(train_MSI_dir))
#
# fnames <- paste0("MSI.",9001:10500,".jpg")
# file.copy(file.path(original_dataset_dir_MSI,fnames),
#           file.path(validation_MSI_dir))
#
# fnames <- paste0("MSI.",11001:12500,".jpg")
# file.copy(file.path(original_dataset_dir_MSI,fnames),
#           file.path(test_MSI_dir))
#
# ############ MSS 5001-12500  ############
# fnames <- paste0("MSS.",5001:8000,".jpg")
# file.copy(file.path(original_dataset_dir_MSS,fnames),
#           file.path(train_MSS_dir))
#
# fnames <- paste0("MSS.",9001:10500,".jpg")
# file.copy(file.path(original_dataset_dir_MSS,fnames),
#           file.path(validation_MSS_dir))
#
# fnames <- paste0("MSS.",11001:12500,".jpg")
# file.copy(file.path(original_dataset_dir_MSS,fnames),
#           file.path(test_MSS_dir))

```
```
```

```
# VGG16

Plases see below the VGG16 model.
```{r}
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(64,64,3)
)
conv_base
```
Model 5.1 and Model 5.2 Feature extraction
```{r}
# datagen <- image_data_generator(rescale = 1/255)
# batch_size <- 20
#
# extract_features <- function(directory,sample_count){
#   features <- array(0,dim = c(sample_count,4,4,512))
#   labels <- array(0,dim = c(sample_count))
#
#   generator <- flow_images_from_directory(
#     directory = directory,
#     generator = datagen,
#     target_size = c(64,64),
#     batch_size = batch_size,
#     class_mode = "binary"
#   )
#   i <- 0
#   while(TURE){
#     batch <- generator_next(generator)
#     inputs_batch <- batch[[1]]
#     labels_batch <- batch[[2]]
#     features_batch <- conv_base %>% oredict(inputs_batch)
#
#     index_range <- ((i * batch_size)+1):((i+1)*batch_size)
#     features[index_range,,,] <- features_batch
#     labels[index_range] <- labels_batch
#
#     i <- i+1
#     if(i*batch_size >=sample_count)
#       break
#   }
#   list(
#     features=features,
#     labels = labels
#   )
# }
# train <- extract_features(train_dir,2000)
# validation <- extract_features(validation_dir,1000)
# test <- extract_features(test_dir,1000)
train_dataset <-image_dataset_from_directory(train_dir,
```
```

```
                                            image_size = c(64, 64),
                                            batch_size = 32)
validation_dataset <- image_dataset_from_directory(validation_dir,
                                                   image_size = c(64, 64),
                                                   batch_size = 32)
test_dataset <- image_dataset_from_directory(test_dir,
                                            image_size = c(64, 64),
                                            batch_size = 32)

get_features_and_labels <- function(dataset) {
  n_batches <- length(dataset)
  all_features <- vector("list", n_batches)
  all_labels <- vector("list", n_batches)
  iterator <- as_array_iterator(dataset)
  for (i in 1:n_batches) {
    c(images, labels) %<-% iter_next(iterator)
    preprocessed_images <- imagenet_preprocess_input(images)
    features <- conv_base %>% predict(preprocessed_images)
    all_labels[[i]] <- labels
    all_features[[i]] <- features
    }
  all_features <- listarrays::bind_on_rows(all_features)
  all_labels <- listarrays::bind_on_rows(all_labels)
  list(all_features, all_labels)
  }
c(train_features, train_labels) %<-% get_features_and_labels(train_dataset)
c(val_features, val_labels) %<-% get_features_and_labels(validation_dataset)
c(test_features, test_labels) %<-% get_features_and_labels(test_dataset)

dim(train_features)

inputs <- layer_input(shape = c(2, 2, 512))
outputs <- inputs %>%
  layer_flatten() %>%
  layer_dense(256) %>%
  layer_dropout(.5) %>%
  layer_dense(1, activation = "sigmoid")
model5.1.1 <- keras_model(inputs, outputs)
model5.1.1 %>% compile(loss = "binary_crossentropy",
                       optimizer = "rmsprop",
                       metrics = "accuracy")
summary(model5.1.1)
callbacks <- list(
  callback_model_checkpoint(
    filepath = "feature_extraction.keras",
    save_best_only = TRUE,
    monitor = "val_loss"
    )
  )
history5.1.1 <- model5.1.1 %>% fit(
  train_features, train_labels,
```

```
  epochs = 50,
  validation_data = list(val_features, val_labels),
  callbacks = callbacks
  )

plot(history5.1.1)
model5.1.1 %>% evaluate_generator(test_generator,steps = 80)
```


```{r}
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(64,64,3)
)

freeze_weights(conv_base)

unfreeze_weights(conv_base)
cat("This is the number of trainable weights before freezing,",
    "the conv base:", length(conv_base$trainable_weights),"\n")
freeze_weights(conv_base)
cat("This is the number of trainable weights after freezing,",
    "the conv base:", length(conv_base$trainable_weights),"\n")

# data_augmentation <- keras_model_sequential() %>%
#   layer_random_flip("horizontal") %>%
#   layer_random_rotation(0.1) %>%
#   layer_random_zoom(0.2)
# inputs <- layer_input(shape = c(64, 64, 3))
# outputs <- inputs %>%
#   data_augmentation() %>%
#   imagenet_preprocess_input() %>%
#   conv_base() %>%
#   layer_flatten() %>%
#   layer_dense(256) %>%
#   layer_dropout(0.5) %>%
#   layer_dense(1, activation = "sigmoid")

model5.2.1 <- keras_model_sequential() %>%
  conv_base %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
model5.2.1
```



```{r}
```

```
train_datagen <- image_data_generator(
  rescale = 1/255,
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = TRUE,
  fill_mode = "nearest"
)

test_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory (train_dir,
                                               train_datagen,
                                               target_size = c(64, 64),
                                               batch_size = 80,#80
                                               class_mode = "binary"
                                               )
validation_generator <- flow_images_from_directory (validation_dir,
                                                    test_datagen,
                                                    target_size = c(64, 64),
                                                    batch_size = 80,#80
                                                    class_mode = "binary")

model5.2.1 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 2e-5),
  metrics = c("accuracy")
)

history5.2.1 <- model5.2.1 %>% fit (
  train_generator,
  steps_per_epoch = 100,
  epochs = 50,  # 20
  validation_data = validation_generator,
  validation_steps = 50)

plot(history5.2.1)
model5.2.1 %>% evaluate_generator(test_generator,steps = 80)
```

Model 5.3 Fine tuning

```{r}
conv_base
freeze_weights(conv_base)
unfreeze_weights(conv_base,from = -4)

model5.3.1 <- keras_model_sequential() %>%
  conv_base %>%
```

```
  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
summary(model5.3.1)


model5.3.1 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-5),
  metrics = c("accuracy")
)

history5.3.1 <- model5.3.1 %>% fit (
  train_generator,
  steps_per_epoch = 100,
  epochs = 50,  # 20
  validation_data = validation_generator,
  validation_steps = 50)

plot(history5.3.1)
model5.3.1 %>% evaluate_generator(test_generator,steps = 80)

```
Model 5.4 Run from end to end
```{r}
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(64,64,3)
)
conv_base


model5.4.1 <- keras_model_sequential() %>%
  conv_base %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
summary(model5.4.1)


model5.4.1 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-5),
  metrics = c("accuracy")
)

history5.4.1 <- model5.4.1 %>% fit (
  train_generator,
  steps_per_epoch = 100,
  epochs = 50 ,  # 20
```

```
  validation_data = validation_generator,
  validation_steps = 50)

plot(history5.4.1)

history5.4.2 <- model5.4.1 %>% fit (
  train_generator,
  steps_per_epoch = 100,
  epochs = 100 ,  # 20
  validation_data = validation_generator,
  validation_steps = 50)

plot(history5.4.2)
test_generator <- flow_images_from_directory(
  test_dir,
  test_datagen,
  target_size = c(64,64),
  batch_size = 50,
  class_mode = "binary"
)

model5.4.1 %>% evaluate_generator(test_generator,steps = 80)
```

```
# PCA
```

```{r}
flatten_layer_name <- "flatten_3"

flatten_layer_model <- keras_model(inputs = model5.4.1$input, outputs = get_
    layer(model5.4.1, flatten_layer_name)$output)

all_flatten_output <- NULL
all_labels <- NULL

while(TRUE) {
  batch <- generator_next(train_generator)
  input_data <- batch[[1]]
  labels <- batch[[2]]

  if (is.null(all_flatten_output)) {
    all_flatten_output <- predict(flatten_layer_model, input_data)
    all_labels <- labels
  } else {
    all_flatten_output <- rbind(all_flatten_output, predict(flatten_layer_model,
         input_data))
    all_labels <- c(all_labels, labels)
  }
  if (length(all_labels) >= train_generator$samples) {
    break
  }
```

```
}
all_flatten_output <- all_flatten_output[, apply(all_flatten_output, 2, var) !=
    0]

pca_result <- prcomp(all_flatten_output)
plot(pca_result)
a <- summary(pca_result)
pca_data <- data.frame(pca_result$x)
pca_data$label <- as.factor(all_labels)

ggplot(pca_data, aes(x = PC1, y = PC2, color = label)) +
  geom_point() +
  scale_color_manual(values = c("0" = "red", "1" = "black")) +
  theme_minimal() +
  labs(title = "PCA of Flatten_3 Layer Output with Labels",
       x = "PC1",
       y = "PC2") +
  geom_hline(yintercept = 0, color = "blue") +
  geom_vline(xintercept = 0, color = "blue")

```
t-SNE & UMAP
```{r}

#install.packages("umap")
#install.packages("Rtsne")

library(umap)
library(Rtsne)

#t-SNE
tset.seed(123)
tsne_result <- Rtsne(all_flatten_output)
tsne_data <- data.frame(tsne_result$Y)
tsne_data$label <- as.factor(all_labels)

ggplot(tsne_data, aes(x = X1, y = X2, color = label)) +
  geom_point() +
  scale_color_manual(values = c("0" = "red", "1" = "black")) +
  theme_minimal() +
  labs(title = "t-SNE of Flatten_3 Layer Output with Labels",
       x = "t-SNE 1",
       y = "t-SNE 2") +
  geom_hline(yintercept = 0, color = "blue") +
  geom_vline(xintercept = 0, color = "blue")

#UMAP
umap_result <- umap(all_flatten_output)
umap_data <- data.frame(umap_result$layout)
umap_data$label <- as.factor(all_labels)
```

```
ggplot(umap_data, aes(x = X1, y = X2, color = label)) +
  geom_point() +
  scale_color_manual(values = c("0" = "red", "1" = "black")) +
  theme_minimal() +
  labs(title = "UMAP of Flatten_3 Layer Output with Labels",
       x = "UMAP 1",
       y = "UMAP 2") +
  geom_hline(yintercept = 0, color = "blue") +
  geom_vline(xintercept = 0, color = "blue")
```

```
# Cats and dogs
```{r}
original_dataset_dir<- "~/TFG/original"
base_dir <- "~/TFG/dog_cat"
#dir.create(base_dir)

train_dir<-file.path(base_dir,"train",fsep ="/")
#dir.create(train_dir)

validation_dir<-file.path(base_dir,"validation",fsep ="/")
#dir.create(validation_dir)

test_dir<-file.path(base_dir,"test",fsep ="/")
#dir.create(test_dir)

train_cats_dir<-file.path(train_dir,"cats",fsep ="/")
#dir.create(train_cats_dir)
train_dogs_dir<-file.path(train_dir,"dogs",fsep ="/")
#dir.create(train_dogs_dir)

validation_cats_dir<-file.path(validation_dir,"cats",fsep ="/")
#dir.create(validation_cats_dir)
validation_dogs_dir<-file.path(validation_dir,"dogs",fsep ="/")
#dir.create(validation_dogs_dir)

test_cats_dir<-file.path(test_dir,"cats",fsep ="/")
#dir.create(test_cats_dir)
test_dogs_dir<-file.path(test_dir,"dogs",fsep ="/")
#dir.create(test_dogs_dir)

# images_dir <- "~/TFG/cats_small/cats_small"
# files <- list.files(images_dir, pattern = "\\.jpg$", full.names = FALSE)
# num_files <- length(files)
# new_names <- sprintf("cats.%d.jpg", 1:num_files)
# # print(new_names)
#
# if (num_files > 0) {
#     new_names <- sprintf("cats.%d.jpg", 1:num_files)
#     new_files <- file.path(images_dir, new_names)
```
```

```
#     result <- mapply(file.rename, from = file.path(images_dir, files), to =
    new_files)
#     print(result)
# } else {
#     print("No JPG files found in the specified directory.")
# }
#
#
# images2_dir <- "~/TFG/cats_small/dogs_small"
# files2 <- list.files(images2_dir, pattern = "\\.jpg$", full.names = FALSE)
# num_files2 <- length(files2)
# new_names2 <- sprintf("dogs.%d.jpg", 1:num_files2)
# #print(new_names2)
# if (num_files2 > 0) {
#     new_names2 <- sprintf("dogs.%d.jpg", 1:num_files2)
#     new_files2 <- file.path(images2_dir, new_names2)
#     result <- mapply(file.rename, from = file.path(images2_dir, files2), to =
    new_files2)
#     print(result)
# } else {
#     print("No JPG files found in the specified directory.")
# }

# original_dataset_dir_cats <- "~/TFG/cats_small/cats_small"
# original_dataset_dir_dogs <- "~/TFG/cats_small/dogs_small"
#
# fnames <- paste0("cat.",1:1000,".jpg")
# file.copy(file.path(original_dataset_dir,fnames),
#           file.path(train_cats_dir))
# #
# fnames <- paste0("cat.",1001:1500,".jpg")
# file.copy(file.path(original_dataset_dir,fnames),
#           file.path(validation_cats_dir))
#
# fnames <- paste0("cat.",1500:2000,".jpg")
# file.copy(file.path(original_dataset_dir,fnames),
#           file.path(test_cats_dir))
#
#  fnames <- paste0("dog.",1:1000,".jpg")
#  file.copy(file.path(original_dataset_dir,fnames),
#            file.path(train_dogs_dir))
#
#  fnames <- paste0("dog.",1001:1500,".jpg")
#  file.copy(file.path(original_dataset_dir,fnames),
#            file.path(validation_dogs_dir))
#
#  fnames <- paste0("dog.",1500:2000,".jpg")
#  file.copy(file.path(original_dataset_dir,fnames),
#            file.path(test_dogs_dir))

```
```
```

```
## VGG16
Plases see below the VGG16 model.
```{r}
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(64,64,3)
)
conv_base
```
Model 5.1 and Model 5.2 Feature extraction
```{r}
# datagen <- image_data_generator(rescale = 1/255)
# batch_size <- 20
#
# extract_features <- function(directory,sample_count){
#   features <- array(0,dim = c(sample_count,4,4,512))
#   labels <- array(0,dim = c(sample_count))
#
#   generator <- flow_images_from_directory(
#     directory = directory,
#     generator = datagen,
#     target_size = c(64,64),
#     batch_size = batch_size,
#     class_mode = "binary"
#   )
#   i <- 0
#   while(TURE){
#     batch <- generator_next(generator)
#     inputs_batch <- batch[[1]]
#     labels_batch <- batch[[2]]
#     features_batch <- conv_base %>% oredict(inputs_batch)
#
#     index_range <- ((i * batch_size)+1):((i+1)*batch_size)
#     features[index_range,,,] <- features_batch
#     labels[index_range] <- labels_batch
#
#     i <- i+1
#     if(i*batch_size >=sample_count)
#       break
#   }
#   list(
#     features=features,
#     labels = labels
#   )
# }
# train <- extract_features(train_dir,2000)
# validation <- extract_features(validation_dir,1000)
# test <- extract_features(test_dir,1000)
train_dataset <-image_dataset_from_directory(train_dir,
```
```

```
                                              image_size = c(64, 64),
                                              batch_size = 32)
validation_dataset <- image_dataset_from_directory(validation_dir,
                                                   image_size = c(64, 64),
                                                   batch_size = 32)
test_dataset <- image_dataset_from_directory(test_dir,
                                              image_size = c(64, 64),
                                              batch_size = 32)

get_features_and_labels <- function(dataset) {
  n_batches <- length(dataset)
  all_features <- vector("list", n_batches)
  all_labels <- vector("list", n_batches)
  iterator <- as_array_iterator(dataset)
  for (i in 1:n_batches) {
    c(images, labels) %<-% iter_next(iterator)
    preprocessed_images <- imagenet_preprocess_input(images)
    features <- conv_base %>% predict(preprocessed_images)
    all_labels[[i]] <- labels
    all_features[[i]] <- features
    }
  all_features <- listarrays::bind_on_rows(all_features)
  all_labels <- listarrays::bind_on_rows(all_labels)
  list(all_features, all_labels)
  }
c(train_features, train_labels) %<-% get_features_and_labels(train_dataset)
c(val_features, val_labels) %<-% get_features_and_labels(validation_dataset)
c(test_features, test_labels) %<-% get_features_and_labels(test_dataset)

dim(train_features)

inputs <- layer_input(shape = c(2, 2, 512))
outputs <- inputs %>%
  layer_flatten() %>%
  layer_dense(256) %>%
  layer_dropout(.5) %>%
  layer_dense(1, activation = "sigmoid")
model5.1 <- keras_model(inputs, outputs)
model5.1 %>% compile(loss = "binary_crossentropy",
                     optimizer = "rmsprop",
                     metrics = "accuracy")
callbacks <- list(
  callback_model_checkpoint(
    filepath = "feature_extraction.keras",
    save_best_only = TRUE,
    monitor = "val_loss"
    )
  )
history5.1 <- model5.1 %>% fit(
  train_features, train_labels,
  epochs = 30,
```

```
  validation_data = list(val_features, val_labels),
  callbacks = callbacks
  )

plot(history5.1)
```


```{r}
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(64,64,3)
)

freeze_weights(conv_base)

unfreeze_weights(conv_base)
cat("This is the number of trainable weights before freezing,",
    "the conv base:", length(conv_base$trainable_weights),"\n")
freeze_weights(conv_base)
cat("This is the number of trainable weights after freezing,",
    "the conv base:", length(conv_base$trainable_weights),"\n")

# data_augmentation <- keras_model_sequential() %>%
#   layer_random_flip("horizontal") %>%
#   layer_random_rotation(0.1) %>%
#   layer_random_zoom(0.2)
# inputs <- layer_input(shape = c(64, 64, 3))
# outputs <- inputs %>%
#   data_augmentation() %>%
#   imagenet_preprocess_input() %>%
#   conv_base() %>%
#   layer_flatten() %>%
#   layer_dense(256) %>%
#   layer_dropout(0.5) %>%
#   layer_dense(1, activation = "sigmoid")

model5.2 <- keras_model_sequential() %>%
  conv_base %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
model5.2
```



```{r}
train_datagen <- image_data_generator(
  rescale = 1/255,
```

```
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = TRUE,
  fill_mode = "nearest"
)

test_datagen <- image_data_generator(rescale = 1/255)

train_generator <- flow_images_from_directory (train_dir,
                                               train_datagen,
                                               target_size = c(64, 64),
                                               batch_size = 20,
                                               class_mode = "binary"
                                               )
validation_generator <- flow_images_from_directory (validation_dir,
                                                    test_datagen,
                                                    target_size = c(64, 64),
                                                    batch_size = 20,
                                                    class_mode = "binary")

model5.2 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 2e-5),
  metrics = c("accuracy")
)

history5.2 <- model5.2 %>% fit (
  train_generator,
  steps_per_epoch = 100,
  epochs = 30,  # 20
  validation_data = validation_generator,
  validation_steps = 50)

```


```{r}
plot(history5.2)
```
Model 5.3 Fine tuning

```{r}
conv_base
freeze_weights(conv_base)
unfreeze_weights(conv_base,from = -4)

model5.3 <- keras_model_sequential() %>%
  conv_base %>%
```

```
  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
model5.3


model5.3 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-5),
  metrics = c("accuracy")
)

history5.3 <- model5.3 %>% fit (
  train_generator,
  steps_per_epoch = 100,
  epochs = 30,  # 20
  validation_data = validation_generator,
  validation_steps = 50)

```
```{r}
plot(history5.3)
```
###Model 5.4 Run from end to end
```{r}
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(64,64,3)
)
conv_base


model5.4 <- keras_model_sequential() %>%
  conv_base %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
model5.4


model5.4 %>% compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(learning_rate = 1e-5),
  metrics = c("accuracy")
)

history5.4 <- model5.4 %>% fit (
  train_generator,
  steps_per_epoch = 100,
  epochs = 30,  # 20
```

```
  validation_data = validation_generator,
  validation_steps = 50)

plot(history5.4)
```

Listing 3: Codes uesd for this paper