UNIVERSITY PARTNER

UNIVERSITY OF WOLVERHAMPTON

HERALD COLLEGE KATHMANDU

Artificial Intelligence and Machine Learning

(6CS012)

Image Classification Task

Butterfly Image Classification

Student Id	: 2227486
Student Name	: Nayan Raj Khanal
Group	: L5CG4
Tutor	: Ms. Sunita Parajuli
Lecturer	: Mr. Siman Giri
Date	: 12th May, 2024

Abstract

In this report three models are assessed for butterfly image classification using the "Butterfly Image Classification" dataset featuring 75 different classes of Butterflies and contains 1000+ labelled images. We first understand the data by organizing the images into appropriate directories and splitting the training and validation data in the ratio of 80:20. Then we checked for corrupted images, created dataset for train, validation and test from image files, visualized the data by printing out number of images per class and applied data augmentation to each element. Following that we built our three models, FCN, CNN and transfer learning with fine-tuning. FCN struggled with capturing spatial information and showed poor accuracy (19.47%) and high loss (3.88%), on the other hand CNN showed improved performance with 85% accuracy and 58% loss but transfer learning with fine-tuning using the ResNet152V2 pre-trained model outperformed both with highest accuracy (91.50%) and loss (34.93%). From these results we can conclude that leveraging pre-trained models and fine-tuning yields increased efficiency and achieves optimal performance.

Table of Contents

Table of Figures

1 Introduction

The dataset of choice is “Butterfly Image Classification”. It features 75 different classes of Butterflies and contains 1000+ labelled images. The goal here is to develop a model that can accurately classify a butterfly into their respective species from an image of a butterfly. (Depie, 2024)

The main aim of this task is to develop an image classification model that can accurately identify butterfly species from their respective image. The objectives of the task are:

- Preprocessing and understanding the dataset
- Implementing various deep learning models for image classification
- Evaluate and comparing the performance of these models
- Fine-tuning the model to improve its accuracy further

In ‘Data Understanding’ task, we performed Data Analysis, Visualization and Pre-Processing. Firstly, we created and defined kaggle directories for train, validation, test and split the training and validation data in the ratio of 80:20. We then plotted a graph containing image of all 75 classes, then checked for corrupted images and print if found any. After that we used image_dataset_from_directory function from keras to generate a dataset for train, validation and test from image files in a directory. We then print out number of images per class. Following that we applied data augmentation to each element to increase our image sample by flipping, rotating, translation and zooming and plotted the graph of augmented images.

Three main models were considered:

- Fully Connected Neural Network (FCN): It's implemented using dense layers in a sequential model architecture, where the output of each layer is connected to the input of the next layer.
- Convolutional Neural Network (CNN): It's implemented using convolutional layers and pooling layers in a sequential model architecture, with additional fully connected layers for classification.
- Transfer Learning with a pre-trained model (ResNet152V2): It involves loading a pre-trained model ResNet, and then adding custom layers on top for fine-tuning.

Each model will be trained and evaluated to determine its performance in classifying butterfly images accurately.

The report includes an introduction, methodology for three models alongside the detailed explanation of models architecture, training processes, findings and final discussions comparing the performance of all models.

2 Methodology

2.1 Image Classification with Fully Connected Neural Network.

Model Summary:

For the first model in our Image Classification task we built a FCN model using dense layers in sequence, each layers output is the input of the next layer. Below is the model's summary:

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 180, 180, 3)	0
flatten (Flatten)	(None, 97200)	0
dense (Dense)	(None, 256)	24,883,456
batch_normalization (BatchNormalization)	(None, 256)	1,024
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 128)	32,896
batch_normalization_1 (BatchNormalization)	(None, 128)	512
dropout_1 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 75)	9,675

Total params: 24,927,563 (95.09 MB)
Trainable params: 24,926,795 (95.09 MB)
Non-trainable params: 768 (3.00 KB)

Figure 1. Model Summary for FCN

The model has a total of 9 layers:

- Input_Layer: Receives input with a shape of (180, 180, 3), representing an image with dimensions 180x180 pixels and 3 color channels (RGB).
- Flatten layer: Flattens the input into a one-dimensional array to feed the dense layers.

- Two Dense layer with ReLU activation (256, 128 units): These layers perform feature extraction and transformation.
- Two Batch_Normalization layer: These layers help stabilize and speed up the training process by normalizing the input activations.
- Two Dropout layer (0.5): These layers help overfitting by randomly setting 50% of the input units as 0.
- Dense layer with softmax activation (Output layer): This layer generates output probabilities for each class (total of 75 classes).

(Tung, 2021) (wcvanvan, 2023)

```
[ ]  def make_fcn_model(input_shape, num_classes):
       '''
       This function defines a fully connected neural network model. It takes
       input_shape and num_classes as arguments and returns a Keras model.
       '''
         # Define input layer
         inputs = Input(shape=input_shape)
         # Flatten
         x = Flatten()(inputs)

         # Dense layer with 256 units and ReLU activation function
         x = Dense(256, activation='relu')(x)
         # Batch normalization layer
         x = BatchNormalization()(x)
         # Dropout layer
         x = Dropout(0.5)(x)

         # Dense layer with 128 units and ReLU activation function
         x = Dense(128, activation='relu')(x)
         # Batch normalization layer
         x = BatchNormalization()(x)
         # Dropout layer
         x = Dropout(0.5)(x)

         # Output layer with softmax activation function
         outputs = Dense(num_classes, activation='softmax')(x)

         # Define the model
         return Model(inputs, outputs)

     # Create the FCN model
     model = make_fcn_model(input_shape=(180, 180, 3), num_classes=75)

     # Print model summary
     model.summary()
```

Figure 2. Code Block for FCN

This function defines a fully connected neural network model. It takes input_shape and num_classes as arguments and returns a Keras model.

The model hyperparameters in the architecture are:

- Number of Units in Dense Layers
- Dropout Rate
- Batch Normalization
- Activation Functions
- Optimizer
- Loss Function
- Metrics
- Callbacks

Training of the Model:

Sparse categorical cross-entropy loss function is used to train the model. Here, the labels are integers rather than vectors. This loss function is suitable for multi-class classification tasks where the classes are mutually exclusive i.e. each data point belongs to exactly one class and not to any other class. (Rahman, 2023)

The RMSprop optimizer is used to train the model. RMSprop is an adaptive learning rate optimization algorithm that doesn't treat learning rate as a hyperparameter, which means the learning rate changes over time. (Sanghvirajit, 2021)

The model was trained for 50 epochs and below is the Train VS Validation Accuracy & Loss graph:

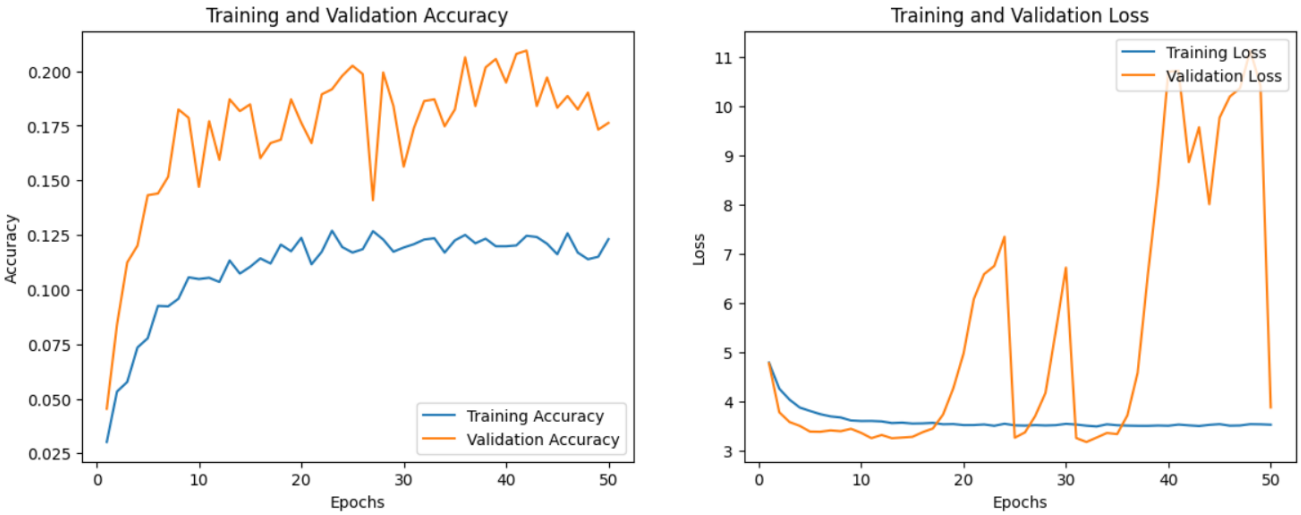

Figure 3. FCN's Train VS Validation Accuracy & Loss

```python
# Get training and validation accuracy
train_acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

# Get training and validation loss
train_loss = history.history['loss']
val_loss = history.history['val_loss']

# Create a range of epochs
epochs_range = range(1, epochs + 1)

# Plot training and validation accuracy
plt.figure(figsize=(14, 5))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, train_acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(epochs_range, train_loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.show()
```

Figure 4. FCN Graph's Code Block

The graph shows the training and validation accuracy and loss curves after training the model. From the graph we can see that:

- Both the training and validation accuracy are low and it is not improving even after training for many epochs.
- The validation loss remains high and does not decrease much during training.
- There is a significant gap between the training and validation performance metrics.

Findings:

```
# Load the saved model
model = tf.keras.models.load_model("/kaggle/working/fcn.keras")

# Evaluate the model on the test dataset
test_loss, test_accuracy = model.evaluate(test_ds)

print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

```
12/12 ━━━━━━━━━━━━━━━━━━━━ 2s 55ms/step - accuracy: 0.2082 - loss: 3.4377
Test Loss: 3.8808255195617676
Test Accuracy: 0.19466666877269745
```

Figure 5. FCN's Evaluation

The test accuracy obtained by the FCN model is approximately 19.47%, with a test loss of approximately 3.88. From this we can infer, the model's performance on the unseen test data is awfully poor. This significant gap between training and test accuracy suggests that the model maybe overfitting and underfitting.

Based on the accuracy it is safe to say that the model struggles to generalize, that means the model's predictions on new, unseen images is not reliable.

Discussions:

The FCN model shows low test accuracy and high test loss. From this we can infer that the model is overfitting and underfitting and the reason for it can be various such as architecture, optimizer, loss function. To address this, techniques such as regularization, dropout, or early stopping can be applied or considering alternative model architectures may improve model's performance.

2.2 Image Classification with Convolutional Neural Network

Model Summary:

For the second model in our Image Classification task we built a CNN model using convolutional layers and pooling layers in a sequence alongside with additional fully connected layers for classification.

Below is the model's summary:

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 180, 180, 3)	0
conv2d (Conv2D)	(None, 178, 178, 32)	896
max_pooling2d (MaxPooling2D)	(None, 89, 89, 32)	0
batch_normalization (BatchNormalization)	(None, 89, 89, 32)	128
dropout (Dropout)	(None, 89, 89, 32)	0
conv2d_1 (Conv2D)	(None, 87, 87, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 43, 43, 64)	0
batch_normalization_1 (BatchNormalization)	(None, 43, 43, 64)	256
dropout_1 (Dropout)	(None, 43, 43, 64)	0
conv2d_2 (Conv2D)	(None, 41, 41, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 20, 20, 128)	0
batch_normalization_2 (BatchNormalization)	(None, 20, 20, 128)	512
dropout_2 (Dropout)	(None, 20, 20, 128)	0
conv2d_3 (Conv2D)	(None, 18, 18, 128)	147,584
max_pooling2d_3 (MaxPooling2D)	(None, 9, 9, 128)	0
batch_normalization_3 (BatchNormalization)	(None, 9, 9, 128)	512
dropout_3 (Dropout)	(None, 9, 9, 128)	0

Figure 6. Model Summary for CNN (1)

conv2d_4 (Conv2D)	(None, 7, 7, 128)	147,584
max_pooling2d_4 (MaxPooling2D)	(None, 3, 3, 128)	0
batch_normalization_4 (BatchNormalization)	(None, 3, 3, 128)	512
flatten (Flatten)	(None, 1152)	0
dense (Dense)	(None, 128)	147,584
batch_normalization_5 (BatchNormalization)	(None, 128)	512
dropout_4 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 75)	9,675

Total params: 548,107 (2.09 MB)
Trainable params: 546,891 (2.09 MB)
Non-trainable params: 1,216 (4.75 KB)

Figure 7. Model Summary for CNN (2)

The model has a total of 25 layers, some layers perform the same task as in FCN:

- Input_Layer
- Five Convolutional Layers: These layers perform feature extraction by applying convolutional filters to the input image.
- Five MaxPooling2D Layers: These layers down sample the features obtained from the convolutional layers.
- Six Batch_Normalization Layers
- Five Dropout Layers (0.4): These layers help overfitting by randomly setting 40% of the input units as 0.
- Flatten Layer
- Two Dense Layer

```python
def make_cnn_model(input_shape, num_classes):
  '''
  This function defines a Convolutional Neural Network model. It takes
  input_shape and num_classes as arguments and returns a Keras model.
  '''
    # Define input layer
    inputs = Input(shape=input_shape)
    # Convolutional layers with ReLU activation, max pooling, batch normalization, and dropout
    x = Conv2D(32, (3, 3), activation='relu')(inputs)
    x = MaxPooling2D((2, 2))(x)
    x = BatchNormalization()(x)
    x = Dropout(0.4)(x)
    x = Conv2D(64, (3, 3), activation='relu')(x)
    x = MaxPooling2D((2, 2))(x)
    x = BatchNormalization()(x)
    x = Dropout(0.4)(x)
    x = Conv2D(128, (3, 3), activation='relu')(x)
    x = MaxPooling2D((2, 2))(x)
    x = BatchNormalization()(x)
    x = Dropout(0.4)(x)
    x = Conv2D(128, (3, 3), activation='relu')(x)
    x = MaxPooling2D((2, 2))(x)
    x = BatchNormalization()(x)
    x = Dropout(0.4)(x)
    x = Conv2D(128, (3, 3), activation='relu')(x)
    x = MaxPooling2D((2, 2))(x)
    x = BatchNormalization()(x)
    # Flatten layer, dense layer, batch normalization, and dropout
    x = Flatten()(x)
    x = Dense(128, activation='relu')(x)
    x = BatchNormalization()(x)
    x = Dropout(0.4)(x)
    # Output layer with softmax activation
    outputs = Dense(num_classes, activation='softmax')(x)
    # Define the model
    return Model(inputs, outputs)
# Create the CNN model
model = make_cnn_model(input_shape=(180, 180, 3), num_classes=75)
# Print model summary
model.summary()
```

Figure 8. Code Block for CNN

This function defines a Convolutional Neural Network model. It takes input_shape and num_classes as arguments and returns a Keras model.

The model hyperparameters in the architecture are:

- Number of Units in Dense Layers
- Dropout Rate
- Batch Normalization
- Activation Functions
- Optimizer
- Loss Function
- Metrics
- Callbacks
- Convolutional Filter Size
- Number of Filters
- Number of Filters
- Pooling Size

Training of the Model:

Just like FCN, Sparse categorical cross-entropy loss function and RMSprop optimizer is used to train the model for 120 epochs. Below is the Train VS Validation Accuracy & Loss graph:

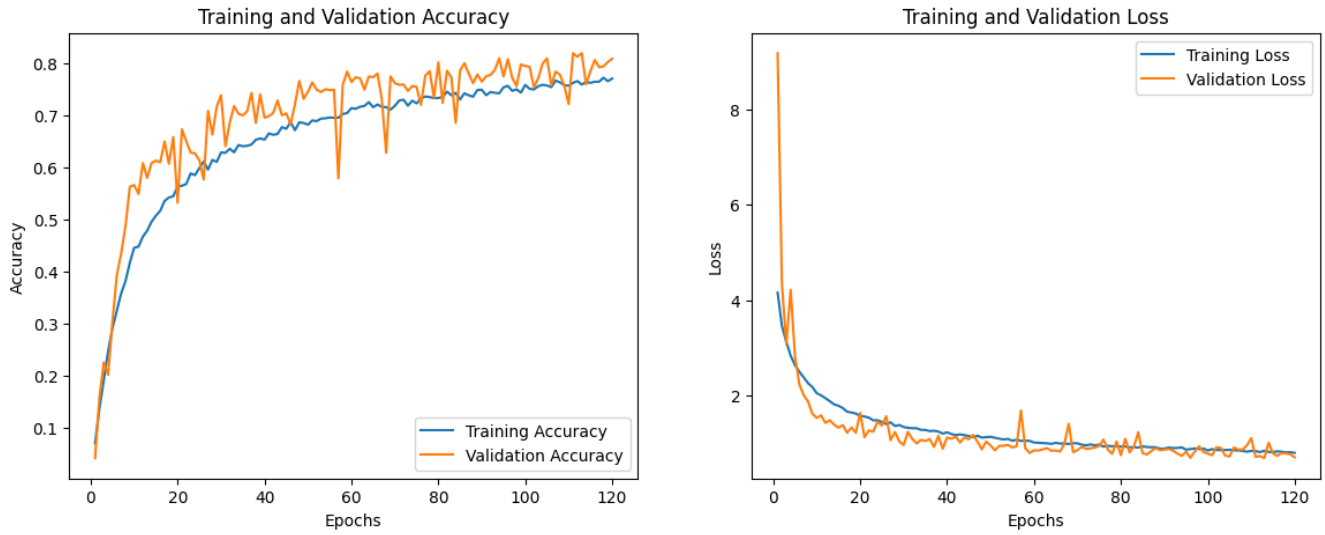

Figure 9. CNN's Train VS Validation Accuracy & Loss

```
# Get training and validation accuracy
train_acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

# Get training and validation loss
train_loss = history.history['loss']
val_loss = history.history['val_loss']

# Create a range of epochs
epochs_range = range(1, epochs + 1)

# Plot training and validation accuracy
plt.figure(figsize=(14, 5))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, train_acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(epochs_range, train_loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.show()
```

Figure 10. CNN Graph's Code Block

Just like FCN the graph shows the training and validation accuracy and loss curves after training the model. From the graph we can see that:

- Both training and validation accuracy are high low and is improving after training for many epochs, given some slight noise on the validation accuracy.
- The validation and training loss are also dropping after training for many epochs, again given some slight noise on the validation loss.
- There is no significant gap between the training and validation performance metrics.

Findings:

```
# Load the saved model
model = tf.keras.models.load_model("/kaggle/working/best_model.keras")

# Evaluate the model on the test dataset
test_loss, test_accuracy = model.evaluate(test_ds)

print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

```
12/12 ━━━━━━━━━━━━━━━━━━━━ 3s 168ms/step - accuracy: 0.8565 - loss: 0.5869
Test Loss: 0.5800039172172546
Test Accuracy: 0.8506666421890259
```

Figure 11. CNN's Evaluation

The model achieved a high accuracy of approximately 85% and test loss of 58% on the test dataset. From this we can infer that the model has learned meaningful patterns from the training data and can generalize well to unseen data. Also, the model's predictions on the test dataset have relatively low error rates.

The CNN model demonstrates high accuracy and relatively low loss indicating effective generalization to unseen data.

Discussions:

The model shows good performance on training and validation data, with high accuracy and low loss curves. However, after 20 epoch the model shows destabilization, we can see it picked up some noises which tells use that the model might be starting to be overfit. Even though it is showing optimal results, the model can still be improved.

2.3 Image Classification with Transfer Learning

Model Summary:

For the third model in our Image Classification task, we employed transfer learning using the ResNet152V2 pre-trained model as the base architecture. Below is the model's summary:

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 180, 180, 3)	0
random_flip (RandomFlip)	(None, 180, 180, 3)	0
random_rotation (RandomRotation)	(None, 180, 180, 3)	0
random_translation (RandomTranslation)	(None, 180, 180, 3)	0
random_zoom (RandomZoom)	(None, 180, 180, 3)	0
true_divide (TrueDivide)	(None, 180, 180, 3)	0
subtract (Subtract)	(None, 180, 180, 3)	0
resnet152v2 (Functional)	(None, 6, 6, 2048)	58,331,648
global_average_pooling2d (GlobalAveragePooling2D)	(None, 2048)	0
dense_1 (Dense)	(None, 2048)	4,196,352
dropout (Dropout)	(None, 2048)	0
dense (Dense)	(None, 75)	153,675

Total params: 62,681,675 (239.11 MB)
Trainable params: 4,350,027 (16.59 MB)
Non-trainable params: 58,331,648 (222.52 MB)
None

The model has its own layers and layers from the base model:

- Input_Layer
- Random Data Augmentation Layers: These layers apply random flip, rotation, translation and zoom to input images.
- Preprocessing Layer: This layer applies specific preprocessing required by the ResNet152V2 model.
- ResNet152V2 Base Model: ResNet152V2 is the base, pre-trained on the ImageNet dataset. It has convolutional layers, batch_normalization layers, activation functions and pooling layers.
- Global_Average_Pooling_Layer: This layer is applied to output feature and reduces the spatial dimension and gets information.
- Dropout_Layer
- Dense layer with softmax activation (Output layer)

The model hyperparameters in the architecture are:

- Learning Rate
- Optimizer
- Loss function
- Callbacks
- Metrics
- Number of Units in Dense Layers
- Dropout Rate

```
[ ]  # Load the ResNet152V2 model pre-trained on ImageNet dataset excluding fully connected layers and using pre-trained weights
     base_model = tf.keras.applications.ResNet152V2(input_shape=IMG_SHAPE,
                                                    include_top=False,
                                                    weights='imagenet'
                                                    )
```

Figure 12. ResNet152V2's Code Block

Here we load the ResNet152V2 model and define it with our input shape excluding fully connected layers as we will add our own dense layers for classification and specify that the pre-trained weights should be used.

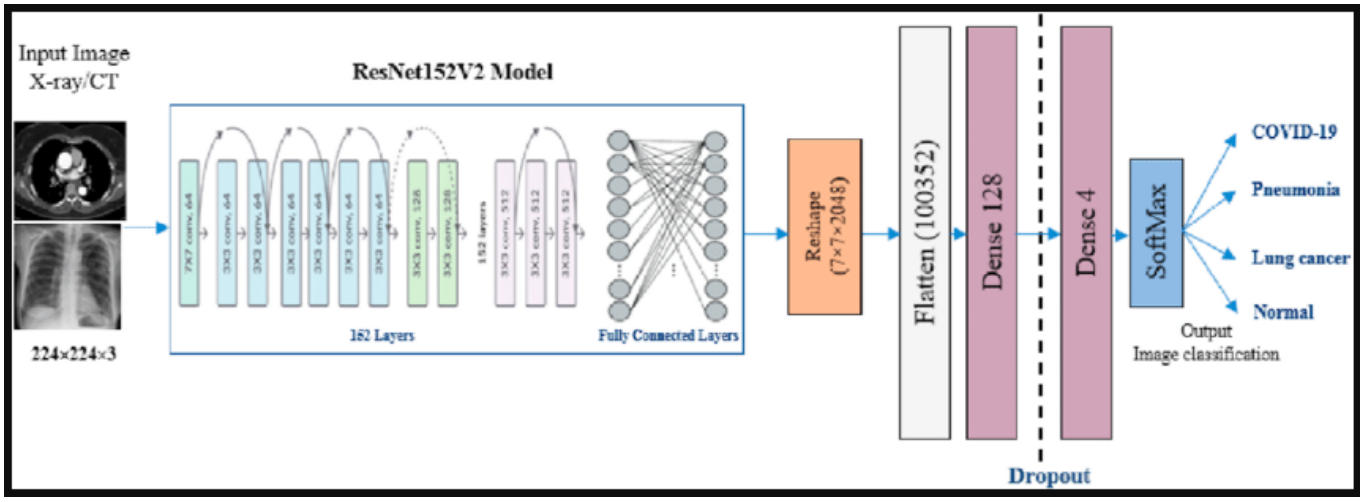

Figure 13. ResNet152V2 Model Architecture

Training of the Model:

From 564 layers of the base model we froze upto 250 layers allowing 314 layers to be trainable. Just like FCN, CNN we used Sparse categorical cross-entropy loss function and RMSprop as the optimizer trained the model for 45 epochs. Below is the Train VS Validation Accuracy & Loss graph:

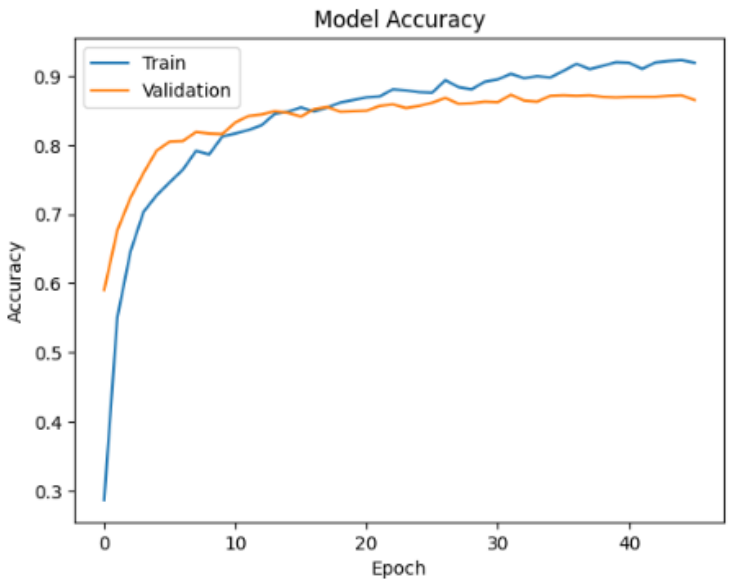

Figure 15. Fine-Tuning Model Accuracy

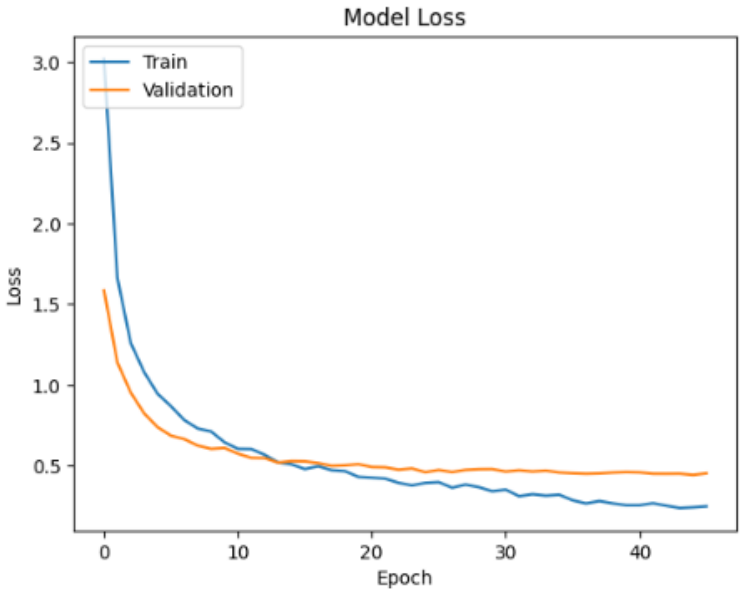

Figure 14. Fine-Tuning Model Loss

Just like FCN and CNN the graph shows the training and validation accuracy and loss curves after training the model. From the graph we can see that:

- Both training and validation accuracy are again high and low and is improving after training for many epochs, given with less to none noise.
- The validation and training loss are also dropping after training for many epochs, again with less to none noise.
- There is minimal gap between the training and validation performance metrics.

Findings:

```
# Load the saved model
model = keras.models.load_model("/kaggle/working/transfer_model_fine.keras")

# Evaluate the model on the test dataset
test_loss, test_accuracy = model.evaluate(test_ds)

print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")
```

```
12/12 ━━━━━━━━━━━━━━━━━━━━ 10s 127ms/step - accuracy: 0.9150 - loss: 0.3031
Test Loss: 0.34941622614860535
Test Accuracy: 0.909333348274231
```

Figure 16. Fine-Tuning's Evaluation

The model achieved a high accuracy of approximately 91% and a test loss of around 35% on the test dataset. From this we can infer that the model has learned meaningful patterns from the training data and can generalize well to unseen data. Also, the predictions on the test dataset have minimal error rates and outperforming both FCN and CNN's predictions.

The transfer learning model demonstrates high accuracy and relatively low loss, indicating effective generalization to unseen data.

We then printed the precision and recall for each class to get insights into the model's performance on individual butterfly species. From the results some classes achieved perfect precision and recall values (eg. ADONIS, AFRICAN GIANT SWALLOWTAIL, AMERICAN SNOOT), whereas (eg. COPPER TAIL, SOUTHERN DOGFACE) had lower precision and recall values.

Discussions:

The transfer learning model showed the best accuracy with minimal loss. The results from both the test and prediction were satisfactory. However, even with high accuracy after interpreting the graph it could be noted that there might be signs of overfitting after a certain number of epochs. If trained for too many epochs the noise in accuracy and loss curves might degrade the performance.

3 Final Discussions

3.1 Image Classification with Fully Connected Neural Network

FCN while simple struggled with capturing spatial information and suffered from overfitting. It results were poor with approximately 19.47% accuracy and 3.88% loss. Evidently the model's failed to generalize to unseen data with high error rates.

3.2 Image Classification with Convolutional Neural Network

CNN, with its convolutional layers performing feature extraction, outperforms FCN in image classification tasks. It results were high with approximately 85% accuracy and 58% loss. The model demonstrated effective generalization to unseen data with low error rates.

3.3 Image Classification with Transfer learning and fine – tuning

Transfer Learning with fine-tuning pre-trained model ResNet152V2 provided the highest accuracy and generalization. The results came out to be approximately 91.50% accuracy and 34.93% loss. The model also demonstrated effective generalization to unseen data with low to none error rates.

Overall, FCN might serve as an entry-level approach for Image Classification Task, CNN result is superior performance and generalization and finally transfer learning with fine-tuning results the best performance and generalization.

4 References

Depie, 2024. *Butterfly Image Classification*. [Online]
Available at: https://www.kaggle.com/datasets/phucthaiv02/butterfly-image-classification/data
[Accessed 1 May 2024].

Rahman, M., 2023. *What You Need to Know about Sparse Categorical Cross Entropy*. [Online]
Available at: https://rmoklesur.medium.com/what-you-need-to-know-about-sparse-categorical-cross-entropy-9f07497e3a6f
[Accessed 3 May 2024].

Sanghvirajit, 2021. *A Complete Guide to Adam and RMSprop Optimizer*. [Online]
Available at: https://medium.com/analytics-vidhya/a-complete-guide-to-adam-and-rmsprop-optimizer-75f4502d83be
[Accessed 3 May 2024].

Tung, T. L., 2021. *Uncover The Beauty Of Image Classification Model With Various Input Resolution*. [Online]
Available at: https://medium.com/vitrox-publication/uncover-the-beauty-of-image-classification-model-with-various-input-resolution-5feeea539964
[Accessed 3 May 2024].

wcvanvan, 2023. *Weight Initialization and Batch Normalization*. [Online]
Available at: https://wcvanvan.medium.com/weight-initialization-for-fcn-d9ea9e0e8064
[Accessed 3 May 2024].